QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 12, 2017

M. Bishop
Microsoft

February 8, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpack-02

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 12, 2017.

Copyright Notice

Table of Contents

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don't live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table

size, which MUST be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Previous versions of QPACK were small deltas of HPACK to introduce order-resiliency. This version departs from HPACK more substantially to add resilience against reset message streams.

In the following sections, this document proposes a new version of HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119] and indicate requirement levels for compliant implementations.

2. QPACK

2.1. Basic model

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into two channels:

- A connection-wide series of table update instructions sent on a dedicated headers stream
- Non-modifying instructions which use the current header table state to encode message headers

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset.

2.2. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541].

The dynamic table is a map from index to header field. Indices are arbitrary numbers greater than the last index of the static table and less than 2^27. Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

The dynamic table is still constrained to the size specified by the decoder. An attempt to add a header to the dynamic table which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can delete entries in the dynamic table, but can only reuse the index or the space after receiving confirmation of a successful deletion.

Because it is possible for QPACK frames to arrive which reference indices which have not yet been defined, such frames MUST wait until another frame has arrived and defined the index. In order to guard against malicious peers, implementations SHOULD impose a time limit and treat expiration of the timer as a decoding error. However, if the implementation chooses not to abort the connection, the remainder of the header block MUST be decoded and the output discarded.

2.2.1. Changes to Header Table Size

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST immediately emit delete instructions which, upon completion, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for these delete instructions to complete.

2.2.2. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur on a dedicated control stream. Message control streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which MUST NOT exceed the declared memory size of the decoder.

The encoder SHOULD track the following information about each entry in the table:

- The list of recently-active streams which reference the entry in a trailer block, if any
- The list of recently-active streams which reference the entry in a non-trailer block, if any

"Recently-active" streams are those which are still open or were closed less than a reasonable number of RTTs ago. An implementation MAY vary its definition of "recent" to trade off memory consumption and timely completion of deletes.

The encoder MUST consider memory as committed beginning when the indexed entry is assigned.

When the encoder wishes to delete an inserted value, it flows through the following set of states:

1. *Delete requested.* The encoder emits a delete instruction indicating which streams might have referenced the entry. The encoder MUST NOT reference the entry in any subsequent frame until this state machine has completed and MUST continue to include the entry in its calculation of consumed memory.
2. *Delete pending.* The decoder receives the delete instruction and checks the current state of its incoming streams (see Section 2.3.2.2). If more references might arrive, it stores the streams still needed and waits for them to complete.
3. *Delete acknowledged.* The decoder has received all QPACK frames which reference the deleted value, and can safely delete the entry. The decoder SHOULD promptly emit a Delete-Ack instruction on the header management stream.
4. *Delete completed.* When the encoder receives a Delete-Ack instruction acknowledging the delete, it no longer counts the

size of the deleted entry against the table size and MAY emit insert instructions for the field with a new value.

2.3. Format of Header Management stream

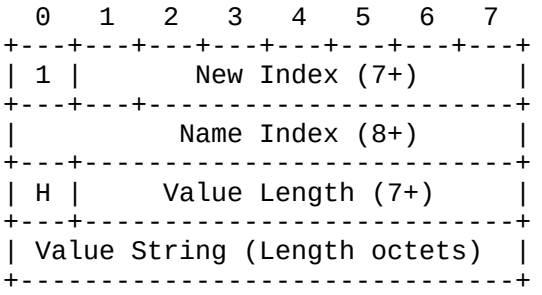
The header management stream contains a series of QPACK instructions with no message boundaries. Data on this stream SHOULD be processed as soon as it arrives.

This section describes the instructions which are possible on the Header Management stream.

2.3.1. Insert

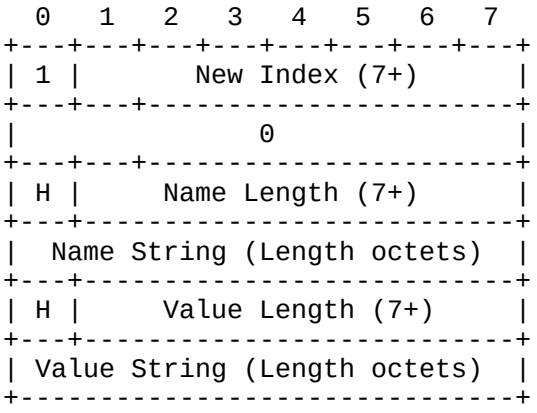
An addition to the header table starts with the '1' one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. This value is always greater than the number of entries in the static table.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with an 8-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Insert Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 8-bit index, followed by the header field name.



Insert Header Field -- New Name

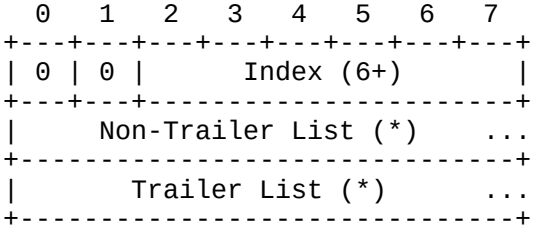
Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder MUST NOT attempt to place a value at an index not known to be vacant. A decoder MUST treat the attempt to insert into an occupied slot as a fatal error.

2.3.2. Delete

A deletion from the header table starts with the ’00’ two bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

A delete instruction then encodes a series of stream IDs which might have contained references to the entry in question.



Delete Instruction

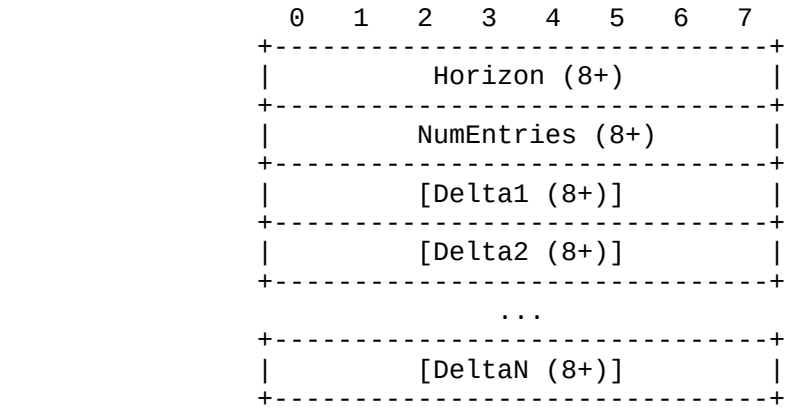
Both the Non-Trailer List and Trailer List are Stream ID Lists (see below) encoding a list of streams which might have referenced the entry either in non-trailer or trailer blocks.

2.3.2.1. Stream ID List

A Stream ID List encodes a sequence of stream IDs in two parts: First, a Horizon value indicates the first non-occurrence about which data is maintained. If data is maintained from the beginning of the connection, the Horizon is zero. This allows senders to succinctly express both old state which has been discarded and large regions where many or all streams contain references.

Following the horizon, a sequence of deltas indicates all streams since the Horizon on which a value has been used.

In the simplest case, a Stream ID List might be a horizon value followed by one zero byte. This indicates an absolute cut-off after which the entry is guaranteed not to be referenced.



Stream ID List

The field are as follows:

Horizon: The ID of the first stream for which the sender retains state which does not reference the deleted entry in the indicated block

NumEntries: The number of streams greater than the Horizon which might reference the entry and are listed in the remainder of the instruction

Delta1..N: A sequence of streams greater than the Horizon which might reference the entry, encoded as the difference in stream number from the previously-listed stream. This field is repeated NumEntries times.

2.3.2.2. Delete Validation

In order to safely delete an entry, a decoder MUST ensure that all outstanding references have arrived and been processed. Because no data is available about stream IDs less than the Horizon, a decoder MUST assume that any earlier stream ID might have contained a reference to the value in question.

A decoder can ensure all outstanding references have been processed by verifying that the following statements are true:

- In the Non-Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of two states:
  - closed
  - headers completely processed
- In the Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of three states:
  - closed
  - headers completely processed AND no trailers are expected
  - trailers completely processed

An implementation MAY omit the "trailers completely processed" case, since the stream is expected to close immediately after receipt of the trailers block.

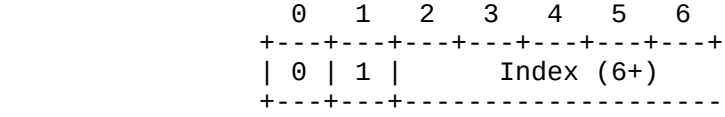
If these conditions are not met upon receipt of a Delete instruction, a decoder MUST wait to emit a Delete-Ack instruction until the outstanding streams have reached an appropriate state.

Note that a decoder MAY condense the list of specified streams by increasing the Horizon value and discarding those explicitly-listed stream IDs which are less than the new Horizon it has chosen. This delays delete completion, but reduces the amount of state to be tracked by the decoder without changing the correctness of the requirements above.

2.3.3. Delete-Ack

Confirmation that a delete has completed is expressed by an instruction which starts with the '01' two-bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

Note that unlike all other instructions, this instruction refers to the receiver's dynamic table, not the sender's.



Delete-Ack Instruction

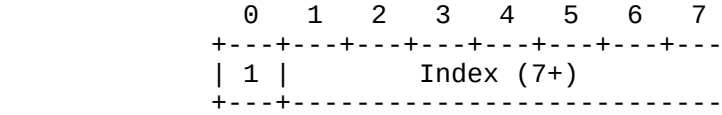
This instruction MUST NOT be sent before the conditions described in Section 2.3.2.2 have been satisfied, and SHOULD be sent as soon as possible once they are.

2.4. Format of Encoded Headers on Message Streams

Frames which carry HTTP message headers encode them using the following instructions:

2.4.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].



Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the index of the matching header field, represented as an integer with a 7-bit prefix (see Section 5.1 of [RFC7541]).

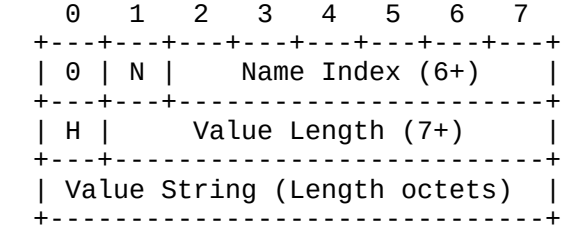
The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

2.4.2. Literal Header Field Representation

A literal header field representation starts with the '0' 1-bit pattern and causes a header field to be added the decoded header list.

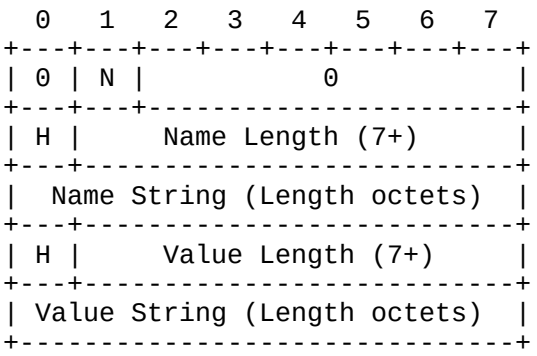
The second bit, 'N', indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the 'N' bit is set, the encoded header MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the 'N' bit set, it MUST use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.



Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2).

3. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, using a Sequence number to enforce ordering. Using QPACK instead would entail the following changes:

- The Sequence field is removed from HEADERS frames (Section 5.2.2) and PUSH_PROMISE frames (Section 5.2.6).
- Header Block Fragments consist of QPACK data instead of HPACK data.
- An additional control stream is reserved for header table updates. Alternately, this could be carried by HEADERS frames on the connection control stream.

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions, but QPACK instructions SHOULD NOT cross a boundary between successive HEADERS frames. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4. Performance Considerations

While QPACK is designed to minimize head-of-line blocking between streams on header decoding, there are some situations in which lost or delayed packets can still impact the performance of header compression.

References to indexed entries will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY proactively insert header values which it believes will be needed on future requests.

Delayed frames which prevent deletes from completing can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to delete entries in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting Delete-Ack instructions to enable the encoder to recover the table space.

5. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder by opening the maximum number of streams, adding entries to the table, then sending delete instructions enumerating many streams in a Stream ID List.

To guard against such attacks, a decoder SHOULD bound its state tracking by generalizing the list of streams to be tracked. This is most easily achieved by advancing the Horizon to a later value and discarding explicit Stream IDs to track, but can also be accomplished by eliding explicit streams in ranges. This does not cause any loss of consistency for deletes, but could delay completion and reduce performance if done aggressively.

6. IANA Considerations

This document currently makes no request of IANA, and might not need to.

7. Acknowledgements

This draft draws heavily on the text of [RFC7541]. The indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- Jana Iyengar
- Patrick McManus

- Martin Thomson
- Charles 'Buck' Krasic
- Kyle Rose

8. Normative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <http://www.rfc-editor.org/info/rfc7541>.

Author's Address

Mike Bishop
Microsoft

Email: michael.bishop@microsoft.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Bishop, Ed.
Microsoft

March 13, 2017

Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-02

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/http .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Table of Contents

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC, drawing heavily on the existing TCP mapping, HTTP/2. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. QUIC Advertisement

An HTTP origin advertises the availability of an equivalent HTTP/QUIC endpoint via the Alt-Svc HTTP response header or the HTTP/2 ALTSVC frame ([RFC7838]), using the ALPN token defined in Section 3.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2 response that HTTP/QUIC was available on UDP port 443 at the same hostname by including the following header in any response:

```
Alt-Svc: hq=":443"
```

On receipt of an Alt-Svc header indicating HTTP/QUIC support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/QUIC clients. QUIC versions are four-octet sequences with no additional constraints on format. Syntax:

```
quic = version-number
version-number = 1*8HEXDIG; hex-encoded QUIC version
```

Leading zeros SHOULD be omitted for brevity. When multiple versions are supported, the "quic" parameter MAY be repeated multiple times in a single Alt-Svc entry. For example, if a server supported both version 0x00000001 and the version rendered in ASCII as "Q034", it could specify the following header:

```
Alt-Svc: hq=":443";quic=1;quic=51303334
```

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Origins SHOULD list only versions which are supported by the alternative, but MAY omit supported versions for any reason.

3. Connection Establishment

HTTP/QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/QUIC support is indicated by selecting the ALPN token "hq" in the crypto handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 5.2.3) MUST be sent as the initial frame of the HTTP control stream (StreamID 3, see Section 4). The server MUST NOT send data on any other stream until the client's SETTINGS frame has been received.

3.1. Draft Version Identification

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Only implementations of the final, published RFC can identify themselves as "hq". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "hq-01".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "hq-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

4. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

QUIC reserves Stream 1 for crypto operations (the handshake, crypto config updates). Stream 3 is reserved for sending and receiving HTTP control frames, and is analogous to HTTP/2's Stream 0. This connection control stream is considered critical to the HTTP connection. If the connection control stream is closed for any reason, this MUST be treated as a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

When HTTP headers and data are sent over QUIC, the QUIC layer handles most of the stream management. An HTTP request/response consumes a pair of streams: This means that the client's first request occurs on QUIC streams 5 and 7, the second on stream 9 and 11, and so on. The server's first push consumes streams 2 and 4. This amounts to the second least-significant bit differentiating the two streams in a request.

The lower-numbered stream is called the message control stream and carries frames related to the request/response, including HEADERS. The higher-numbered stream is the data stream and carries the request/response body with no additional framing. Note that a request or response without a body will cause this stream to be half-closed in the corresponding direction without transferring data.

Because the message control stream contains HPACK data which manipulates connection-level state, the message control stream MUST NOT be closed with a stream-level error. If an implementation chooses to reject a request with a QUIC error code, it MUST trigger a QUIC RST_STREAM on the data stream only. An implementation MAY close (FIN) a message control stream without completing a full HTTP message if the data stream has been abruptly closed. Data on message control streams MUST be fully consumed, or the connection terminated.

All message control streams are considered critical to the HTTP connection. If a message control stream is terminated abruptly for any reason, this MUST be treated as a connection error of type HTTP_RST_CONTROL_STREAM. When a message control stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (see HTTP_MALFORMED_* in Section 6.1).

Pairs of streams must be utilized sequentially, with no gaps. The data stream is opened at the same time as the message control stream is opened and is closed after transferring the body. The data stream is closed immediately after sending the request headers if there is no body.

HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction. Requests and responses are considered complete when the corresponding QUIC streams are closed in the appropriate direction.

4.1. Stream 3: Connection Control Stream

Since most connection-level concerns will be managed by QUIC, the primary use of Stream 3 will be for the SETTINGS frame when the connection opens and for PRIORITY frames subsequently.

4.2. HTTP Message Exchanges

A client sends an HTTP request on a new pair of QUIC streams. A server sends an HTTP response on the same streams as the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 5.2.1) on the control stream containing the message headers (see [RFC7230], Section 3.2),

2. the payload body (see [RFC7230], Section 3.3), sent on the data stream,

3. optionally, one header block on the control stream containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

In addition, prior to sending the message header block indicated above, a response may contain zero or more header blocks on the control stream containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2).

The data stream MUST be half-closed immediately after the transfer of the body. If the message does not contain a body, the corresponding data stream MUST still be half-closed without transferring any data. The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used.

Trailing header fields are carried in an additional header block on the message control stream. Such a header block is a sequence of HEADERS frames with End Header Block set on the last frame. Senders MUST send only one header block in the trailers section; receivers MUST decode any subsequent header blocks in order to maintain HPACK decoder state, but the resulting output MUST be discarded.

An HTTP request/response exchange fully consumes a pair of streams. After sending a request, a client closes the streams for sending; after sending a response, the server closes its streams for sending and the QUIC streams are fully closed.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response and closing its stream. Clients MUST NOT discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

4.2.1. Header Compression

HTTP/QUIC uses HPACK header compression as described in [RFC7541]. HPACK was designed for HTTP/2 with the assumption of in-order delivery such as that provided by TCP. A sequence of encoded header

blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

QUIC streams provide in-order delivery of data sent on those streams, but there are no guarantees about order of delivery between streams. To achieve in-order delivery of HEADERS frames in QUIC, the HPACK-bearing frames contain a counter which can be used to ensure in-order processing. Data (request/response bodies) which arrive out of order are buffered until the corresponding HEADERS arrive.

This does introduce head-of-line blocking: if the packet containing HEADERS for stream N is lost or reordered then the HEADERS for stream N+4 cannot be processed until it has been retransmitted successfully, even though the HEADERS for stream N+4 may have arrived.

DISCUSS: Keep HPACK with HOLB? Redesign HPACK to be order-invariant? How much do we need to retain compatibility with HTTP/2's HPACK?

4.2.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources. In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/QUIC functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The message data stream MUST NOT be closed at the end of the request.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6, on the message control stream.

All QUIC STREAM frames on the message data stream correspond to data sent on the TCP connection. Any QUIC STREAM frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is written to the data stream by the proxy. Note that the

size and number of TCP segments is not guaranteed to map predictably to the size and number of QUIC STREAM frames.

The TCP connection can be closed by either peer. When the client half-closes the data stream, the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will half-close the corresponding data stream. TCP connections which remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT half-close connections on which they are still expecting data.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

4.3. Stream Priorities

HTTP/QUIC uses the priority scheme described in [RFC7540] Section 5.3. In this priority scheme, a given stream can be designated as dependent upon another stream, which expresses the preference that the latter stream (the "parent" stream) be allocated resources before the former stream (the "dependent" stream). Taken together, the dependencies across all streams in a connection form a dependency tree. The structure of the dependency tree changes as PRIORITY frames add, remove, or change the dependency links between streams.

For consistency’s sake, all PRIORITY frames MUST refer to the message control stream of the dependent request, not the data stream.

4.4. Server Push

HTTP/QUIC supports server push as described in [RFC7540]. During connection establishment, the client indicates whether it is willing to receive server pushes via the SETTINGS_DISABLE_PUSH setting in the SETTINGS frame (see Section 3), which defaults to 1 (true).

As with server push for HTTP/2, the server initiates a server push by sending a PUSH_PROMISE frame containing the StreamID of the stream to be pushed, as well as request header fields attributed to the request. The PUSH_PROMISE frame is sent on the control stream of the associated (client-initiated) request, while the Promised Stream ID field specifies the Stream ID of the control stream for the server-initiated request.

The server push response is conveyed in the same way as a non-server-push response, with response headers and (if present) trailers carried by HEADERS frames sent on the control stream, and response body (if any) sent via the corresponding data stream.

5. HTTP Framing Layer

Frames are used only on the connection (stream 3) and message (streams 5, 9, etc.) control streams. Other streams carry data payload and are not framed at the HTTP layer.

This section describes HTTP framing in QUIC and highlights some differences from HTTP/2 framing. For more detail on differences from HTTP/2, see Section 7.1.

5.1. Frame Layout

All frames have the following format:

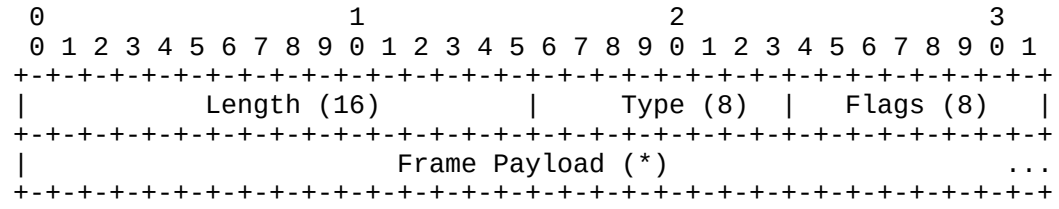


Figure 1: HTTP/QUIC frame format

5.2. Frame Definitions

5.2.1. HEADERS

The HEADERS frame (type=0x1) is used to carry part of a header set, compressed using HPACK [RFC7541].

One flag is defined:

End Header Block (0x4): This frame concludes a header block.

A HEADERS frame with any other flags set MUST be treated as a connection error of type HTTP_MALFORMED_HEADERS.

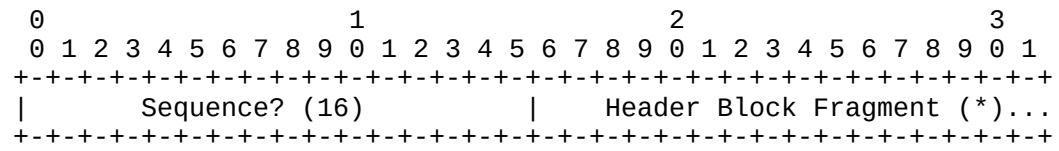


Figure 2: HEADERS frame payload

The HEADERS frame payload has the following fields:

Sequence Number: Present only on the first frame of a header block sequence. This MUST be set to zero on the first header block sequence, and incremented on each header block.

The next frame on the same stream after a HEADERS frame without the EHB flag set MUST be another HEADERS frame. A receiver MUST treat the receipt of any other type of frame as a stream error of type HTTP_INTERRUPTED_HEADERS. (Note that QUIC can intersperse data from other streams between frames, or even during transmission of frames, so multiplexing is not blocked by this requirement.)

A full header block is contained in a sequence of zero or more HEADERS frames without EHB set, followed by a HEADERS frame with EHB set.

On receipt, header blocks (HEADERS, PUSH_PROMISE) MUST be processed by the HPACK decoder in sequence. If a block is missing, all subsequent HPACK frames MUST be held until it arrives, or the connection terminated.

When the Sequence counter reaches its maximum value (0xFFFF), the next increment returns it to zero. An endpoint MUST NOT wrap the Sequence counter to zero until the previous zero-value header block has been confirmed received.

5.2.2. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority of a stream and is substantially different from [RFC7540]. In order to support ordering, it MUST be sent only on the connection control stream. The format has been modified to accommodate not being sent on-stream and the larger stream ID space of QUIC.

The semantics of the Stream Dependency, Weight, and E flag are the same as in HTTP/2.

The flags defined are:

E (0x01): Indicates that the stream dependency is exclusive (see [RFC7540] Section 5.3).

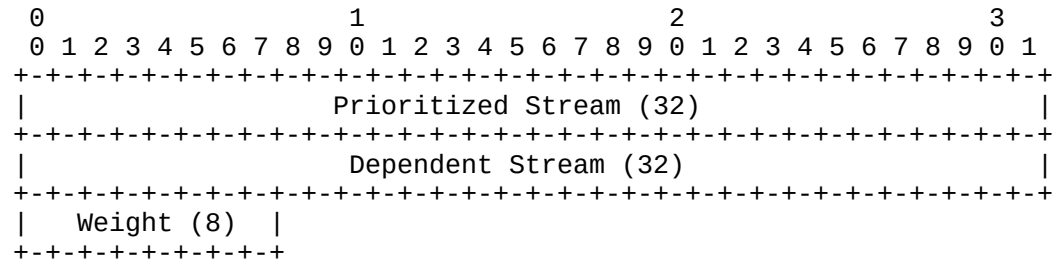


Figure 3: PRIORITY frame payload

The HEADERS frame payload has the following fields:

Prioritized Stream: A 32-bit stream identifier for the message control stream whose priority is being updated.

Stream Dependency: A 32-bit stream identifier for the stream that this stream depends on (see Section 4.3 and {!RFC7540} Section 5.3).

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see [RFC7540] Section 5.3). Add one to the value to obtain a weight between 1 and 256.

A PRIORITY frame MUST have a payload length of nine octets. A PRIORITY frame of any other length MUST be treated as a connection error of type HTTP_MALFORMED_PRIORITY.

5.2.3. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior, and is substantially different from [RFC7540]. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - a peer uses SETTINGS to advertise a set of supported values. The recipient can then choose which entries from this list are also acceptable and proceed with the value it has chosen. (This choice could be announced in a field of an extension frame, or in its own value in SETTINGS.)

Different values for the same parameter can be advertised by each peer. For example, a client might permit a very large HPACK state table while a server chooses to use a small one to conserve memory.

Parameters MUST NOT occur more than once. A receiver MAY treat the presence of the same parameter more than once as a connection error of type HTTP_MALFORMED_SETTINGS.

The SETTINGS frame defines no flags.

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and a length-prefixed binary value.

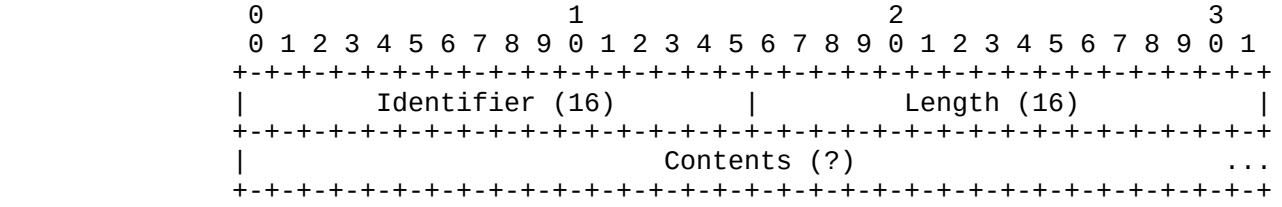


Figure 4: SETTINGS value format

A zero-length content indicates that the setting value is a Boolean and true. False is indicated by the absence of the setting.

Non-zero-length values MUST be compared against the remaining length of the SETTINGS frame. Any value which purports to cross the end of the frame MUST cause the SETTINGS frame to be considered malformed and trigger a connection error of type HTTP_MALFORMED_SETTINGS.

An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

SETTINGS frames always apply to a connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of the connection control stream (see Section 4) by each peer, and MUST NOT be sent subsequently or on any other stream. If an endpoint receives an SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type HTTP_SETTINGS_ON_WRONG_STREAM. If an endpoint receives a second SETTINGS frame, the endpoint MUST respond with a connection error of type HTTP_MULTIPLE_SETTINGS.

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 5.4.1) of type HTTP_MALFORMED_SETTINGS.

5.2.3.1. Integer encoding

Settings which are integers are transmitted in network byte order. Leading zero octets are permitted, but implementations SHOULD use only as many bytes as are needed to represent the value. An integer MUST NOT be represented in more bytes than would be used to transfer the maximum permitted value.

5.2.3.2. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_HEADER_TABLE_SIZE (0x1): An integer with a maximum value of 2^32 - 1.

SETTINGS_DISABLE_PUSH (0x2): Transmitted as a Boolean; replaces SETTINGS_ENABLE_PUSH

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): An integer with a maximum value of 2^32 - 1.

5.2.3.3. Usage in 0-RTT

When a 0-RTT QUIC connection is being used, the client’s initial requests will be sent before the arrival of the server’s SETTINGS frame. Clients SHOULD cache at least the following settings about servers:

- SETTINGS_HEADER_TABLE_SIZE
- SETTINGS_MAX_HEADER_LIST_SIZE

Clients MUST comply with cached settings until the server’s current settings are received. If a client does not have cached values, it SHOULD assume the following values:

- SETTINGS_HEADER_TABLE_SIZE: 0 octets
- SETTINGS_MAX_HEADER_LIST_SIZE: 16,384 octets

Servers MAY continue processing data from clients which exceed its current configuration during the initial flight. In this case, the client MUST apply the new settings immediately upon receipt.

If the connection is closed because these or other constraints were violated during the 0-RTT flight (e.g. with HTTP_HPACK_DECOMPRESSION_FAILED), clients MAY establish a new connection and retry any 0-RTT requests using the settings sent by

the server on the closed connection. (This assumes that only requests that are safe to retry are sent in 0-RTT.) If the connection was closed before the SETTINGS frame was received, clients SHOULD discard any cached values and use the defaults above on the next connection.

5.2.4. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header set from server to client, as in HTTP/2. It defines no flags.

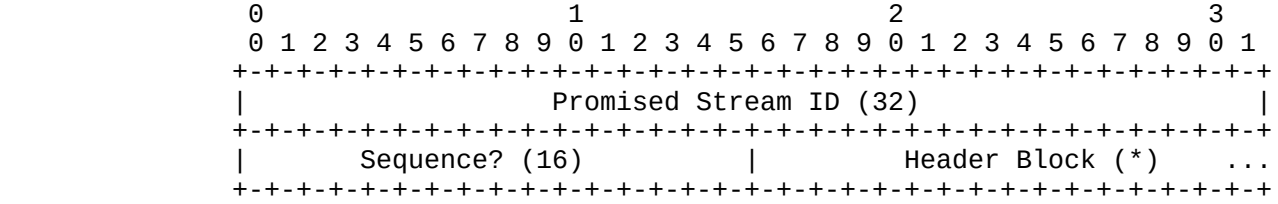


Figure 5: PUSH_PROMISE frame payload

The payload consists of:

Promised Stream ID: A 32-bit Stream ID indicating the QUIC stream on which the response headers will be sent. (The response body stream is implied by the headers stream, as defined in Section 4.)

HPACK Sequence: A sixteen-bit counter, equivalent to the Sequence field in HEADERS

Payload: HPACK-compressed request headers for the promised response.

6. Error Handling

QUIC allows the application to abruptly terminate individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT].

HTTP/QUIC requires that only data streams be terminated abruptly. Terminating a message control stream will result in an error of type HTTP_RST_CONTROL_STREAM.

This section describes HTTP-specific error codes which can be used to express the cause of a connection or stream error.

6.1. HTTP-Defined QUIC Error Codes

QUIC allocates error codes 0x0000-0x3FFF to application protocol definition. The following error codes are defined by HTTP for use in QUIC RST_STREAM, GOAWAY, and CONNECTION_CLOSE frames.

HTTP_PUSH_REFUSED (0x01): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x02): An internal error has occurred in the HTTP stack.

HTTP_PUSH_ALREADY_IN_CACHE (0x03): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x04): The client no longer needs the requested data.

HTTP_HPACK_DECOMPRESSION_FAILED (0x05): HPACK failed to decompress a frame and cannot continue.

HTTP_CONNECT_ERROR (0x06): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x07): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x08): The requested operation cannot be served over HTTP/QUIC. The peer should retry over HTTP/2.

HTTP_MALFORMED_HEADERS (0x09): A HEADERS frame has been received with an invalid format.

HTTP_MALFORMED_PRIORITY (0x0A): A PRIORITY frame has been received with an invalid format.

HTTP_MALFORMED_SETTINGS (0x0B): A SETTINGS frame has been received with an invalid format.

HTTP_MALFORMED_PUSH_PROMISE (0x0C): A PUSH_PROMISE frame has been received with an invalid format.

HTTP_INTERRUPTED_HEADERS (0x0E): A HEADERS frame without the End Header Block flag was followed by a frame other than HEADERS.

HTTP_SETTINGS_ON_WRONG_STREAM (0x0F): A SETTINGS frame was received on a request control stream.

HTTP_MULTIPLE_SETTINGS (0x10): More than one SETTINGS frame was received.

HTTP_RST_CONTROL_STREAM (0x11): A message control stream closed abruptly.

7. Considerations for Transitioning from HTTP/2

HTTP/QUIC is strongly informed by HTTP/2, and bears many similarities. This section points out important differences from HTTP/2 and describes how to map HTTP/2 extensions into HTTP/QUIC.

7.1. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required.

Frame payloads are largely drawn from [RFC7540]. However, QUIC includes many features (e.g. flow control) which are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/QUIC. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/QUIC implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/QUIC will break them.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/QUIC, PRIORITY frames are sent on the connection control stream and the PRIORITY section is removed from the HEADERS frame.

Other than this issue, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with Stream 3 in HTTP/QUIC.

Below is a listing of how each HTTP/2 frame type is mapped:

DATA (0x0): Instead of DATA frames, HTTP/QUIC uses a separate data stream. See Section 4.

HEADERS (0x1): As described above, the PRIORITY region of HEADERS is not supported. A separate PRIORITY frame MUST be used. Padding is not defined in HTTP/QUIC frames. See Section 5.2.1.

PRIORITY (0x2): As described above, the PRIORITY frame is sent on the connection control stream. See Section 5.2.2.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC provides stream lifecycle management.

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 5.2.3 and Section 7.2.

PUSH_PROMISE (0x5): See Section 5.2.4.

PING (0x6): PING frames do not exist, since QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY frames do not exist, since QUIC provides equivalent functionality.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted, and HEADERS frames can be used in series.

The IANA registry of frame types has been updated in Section 9.3 to include references to the definition for each frame type in HTTP/2 and in HTTP/QUIC. Frames not defined as available in HTTP/QUIC SHOULD NOT be sent and SHOULD be ignored as unknown on receipt.

7.2. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/QUIC. The HTTP-level options that are retained in HTTP/QUIC have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See Section 5.2.3.2.

SETTINGS_ENABLE_PUSH: See SETTINGS_DISABLE_PUSH in Section 5.2.3.2.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC requires the maximum number of incoming streams per connection to be specified in the initial transport handshake. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 5.2.3.2.

Settings defined by extensions to HTTP/2 MAY be expressed as integers with a maximum value of 2^32-1, if they are applicable to HTTP/QUIC, but SHOULD have a specification describing their usage. Fields for this purpose have been added to the IANA registry in Section 9.4.

7.3. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, because the error code space is shared between multiple components, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to QUIC error codes as follows:

NO_ERROR (0x0): QUIC_NO_ERROR

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_* error codes defined in Section 6.1.

INTERNAL_ERROR (0x2) HTTP_INTERNAL_ERROR in Section 6.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6) No single mapping. See new error codes defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_HPACK_DECOMPRESSION_FAILED in Section 6.1.

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): HTTP_VERSION_FALLBACK in Section 6.1.

Error codes defined by HTTP/2 extensions need to be re-registered for HTTP/QUIC if still applicable. See Section 9.5.

8. Security Considerations

The security considerations of HTTP over QUIC should be comparable to those of HTTP/2.

The modified SETTINGS format contains nested length elements, which could pose a security risk to an uncautious implementer. A SETTINGS frame parser MUST ensure that the length of the frame exactly matches the length of the settings it contains.

9. IANA Considerations

9.1. Registration of HTTP/QUIC Identification String

This document creates a new registration for the identification of HTTP/QUIC in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "hq" string identifies HTTP/QUIC:

Protocol: HTTP over QUIC

Identification Sequence: 0x68 0x71 ("hq")

Specification: This document

9.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.1

9.3. Existing Frame Types

This document adds two new columns to the "HTTP/2 Frame Type" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the frame type. Values MUST be one of:

* "HTTP/2 only"

* "HTTP/QUIC only"

* "Both"

HTTP/QUIC Specification: Indicates where this frame’s behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Frame Type	Supported Protocols	HTTP/QUIC Specification
DATA	HTTP/2 only	N/A
HEADERS	Both	Section 5.2.1
PRIORITY	Both	Section 5.2.2
RST_STREAM	HTTP/2 only	N/A
SETTINGS	Both	Section 5.2.3
PUSH_PROMISE	Both	Section 5.2.4
PING	HTTP/2 only	N/A
GOAWAY	HTTP/2 only	N/A
WINDOW_UPDATE	HTTP/2 only	N/A
CONTINUATION	HTTP/2 only	N/A

The "Specification" column is renamed to "HTTP/2 specification" and is only required if the frame is supported over HTTP/2.

9.4. Settings Parameters

This document adds two new columns to the "HTTP/2 Settings" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the setting. Values MUST be one of:

* "HTTP/2 only"

* "HTTP/QUIC only"

* "Both"

HTTP/QUIC Specification: Indicates where this setting’s behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Setting Name	Supported Protocols	HTTP/QUIC Specification
HEADER_TABLE_SIZE	Both	Section 5.2.3.2
ENABLE_PUSH / DISABLE_PUSH	Both	Section 5.2.3.2
MAX_CONCURRENT_STREAMS	HTTP/2 Only	N/A
INITIAL_WINDOW_SIZE	HTTP/2 Only	N/A
MAX_FRAME_SIZE	HTTP/2 Only	N/A
MAX_HEADER_LIST_SIZE	Both	Section 5.2.3.2

The "Specification" column is renamed to "HTTP/2 Specification" and is only required if the setting is supported over HTTP/2.

9.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The "HTTP/QUIC Error Code" registry manages a 30-bit space. The "HTTP/QUIC Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 30-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
HTTP_PUSH_REFUSED	0x01	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x02	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x03	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x04	Data no longer needed	Section 6.1
HTTP_HPACK_DECOMPRESSION_FAILED	0x05	HPACK cannot continue	Section 6.1
HTTP_CONNECT_ERROR	0x06	TCP reset or error on CONNECT request	Section 6.1
HTTP_EXCESSIVE_LOAD	0x07	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x08	Retry over HTTP/2	Section 6.1
HTTP_MALFORMED_HEADERS	0x09	Invalid HEADERS frame	Section 6.1
HTTP_MALFORMED_PRIORITY	0x0A	Invalid PRIORITY frame	Section 6.1
HTTP_MALFORMED_SETTINGS	0x0B	Invalid SETTINGS frame	Section 6.1

HTTP_MALFORMED_PUSH_PROMISE	0x0C	Invalid PUSH_PROMISE frame	Section 6.1
HTTP_INTERRUPTED_HEADERS	0x0E	Incomplete HEADERS block	Section 6.1
HTTP_SETTINGS_ON_WRONG_STREAM	0x0F	SETTINGS frame on a request control stream	Section 6.1
HTTP_MULTIPLE_SETTINGS	0x10	Multiple SETTINGS frames	Section 6.1
HTTP_RST_CONTROL_STREAM	0x11	Message control stream was RST	Section 6.1

10. References

10.1. Normative References

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <http://www.rfc-editor.org/info/rfc7541>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <http://www.rfc-editor.org/info/rfc7838>.

10.2. Informative References

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

Appendix A. Contributors

The original authors of this specification were Robbie Shade and Mike Warres.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-01:

- SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS_ACK removed

    * Settings can only occur in the SETTINGS frame a single time

    * Boolean format updated

  - Alt-Svc parameter changed from "v" to "quic"; format updated (#229)

  - Closing the connection control stream or any message control stream is a fatal error (#176)

  - HPACK Sequence counter can wrap (#173)

  - 0-RTT guidance added

  - Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.2. Since draft-ietf-quic-http-00:

  - Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)

  - Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)

  - Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01

  - Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)

  - Described CONNECT pseudo-method (#95)

  - Updated ALPN token and Alt-Svc guidance (#13,#87)

  - Application-layer-defined error codes (#19,#74)

B.3. Since draft-shade-quic-http2-mapping-00:

  - Adopted as base for draft-ietf-quic-http.

  - Updated authors/editors list.

Author’s Address

Mike Bishop (editor)
Microsoft

Email: Michael.Bishop@microsoft.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
I. Swett, Ed.
Google

March 13, 2017

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-02

Abstract

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC implements the spirit of known TCP loss detection mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC loss detection and congestion control, and attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and TCP implementations.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/recovery .

Status of This Memo

Table of Contents

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [QUIC-TRANSPORT].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

This document first describes pre-requisite parts of the QUIC transmission machinery, then discusses QUIC's default congestion control and loss detection mechanisms, and finally lists the various TCP mechanisms that QUIC loss detection implements (in spirit.)

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which makes duplicate detection trivial. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- Retransmittable frames are frames requiring reliable delivery. The most common are STREAM frames, which typically contain application data.

- Crypto handshake data is also sent as STREAM data, and uses the reliability machinery of QUIC underneath.

- ACK frames contain acknowledgment information. QUIC uses a SACK-based scheme, where acks express up to 256 ranges. The ACK frame also includes a receive timestamp for each packet newly acked.

2.1. Relevant Differences Between QUIC and TCP

There are some notable differences between QUIC and TCP which are important for reasoning about the differences between the loss recovery mechanisms employed by the two protocols. We briefly describe these differences below.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet sequence number (referred to as the "packet number") for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with stream offsets encoded within STREAM frames inside of packets that determine delivery order.

QUIC’s packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

2.1.2. No Reneging

QUIC ACKs contain information that is equivalent to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

2.1.3. More ACK Ranges

QUIC supports up to 256 ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery.

2.1.4. Explicit Correction For Delayed Acks

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

We now describe QUIC's loss detection as functions that should be called on packet transmission, when a packet is acked, and timer expiration events.

3.1. Constants of interest

Constants used in loss recovery and congestion control are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxTLPs (default 2): Maximum number of tail loss probes before an RTO fires.

kReorderingThreshold (default 3): Maximum reordering in packet number space before FACK style loss detection considers a packet lost.

kTimeReorderingFraction (default 1/8): Maximum reordering in time sapce before time based loss detection considers a packet lost. In fraction of an RTT.

kMinTLPTimeout (default 10ms): Minimum time in the future a tail loss probe alarm may be set for.

kMinRTOTimeout (default 200ms): Minimum time in the future an RTO alarm may be set for.

kDelayedAckTimeout (default 25ms): The length of the peer's delayed ack timer.

kDefaultInitialRtt (default 100ms): The default RTT used before an RTT sample is taken.

3.2. Variables of interest

We first describe the variables required to implement the loss detection mechanisms described in this section.

loss_detection_alarm: Multi-modal alarm used for loss detection.

handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.

tlp_count: The number of times a tail loss probe has been sent without receiving an ack.

rto_count: The number of times an rto has been sent without receiving an ack.

smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC6298]

rttvar: The RTT variance, computed as described in [RFC6298]

initial_rtt: The initial RTT used before any RTT measurements have been made.

reordering_threshold: The largest delta between the largest acked retransmittable packet and a packet containing retransmittable frames before it’s declared lost.

time_reordering_fraction: The reordering window as a fraction of max(smoothed_rtt, latest_rtt).

loss_time: The time at which the next packet will be considered lost based on early transmit or exceeding the reordering window in time.

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, and a bytes field indicating the packet’s size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost.

3.3. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
loss_detection_alarm.reset()
handshake_count = 0
tlp_count = 0
rto_count = 0
if (UsingTimeLossDetection())
  reordering_threshold = infinite
  time_reordering_fraction = kTimeReorderingFraction
else:
  reordering_threshold = kReorderingThreshold
  time_reordering_fraction = infinite
loss_time = 0
smoothed_rtt = 0
rttvar = 0
initial_rtt = kDefaultInitialRtt
```

3.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows:

- packet_number: The packet number of the sent packet.

- is_retransmittble: A boolean that indicates whether the packet contains at least one frame requiring reliable deliver. The retransmittability of various QUIC frames is described in [QUIC-TRANSPORT]. If false, it is still acceptable for an ack to be received for this packet. However, a caller MUST NOT set is_retransmittable to true if an ack is not expected.

- sent_bytes: The number of bytes sent in the packet.

Pseudocode for OnPacketSent follows:

```
OnPacketSent(packet_number, is_retransmittable, sent_bytes):
  sent_packets[packet_number].packet_number = packet_number
  sent_packets[packet_number].time = now
  if is_retransmittable:
    sent_packets[packet_number].bytes = sent_bytes
    SetLossDetectionAlarm()
```

3.5. On Ack Receipt

When an ack is received, it may acknowledge 0 or more packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
OnAckReceived(ack):
  // If the largest acked is newly acked, update the RTT.
  if (sent_packets[ack.largest_acked]):
    rtt_sample = now - sent_packets[ack.largest_acked].time
    if (rtt_sample > ack.ack_delay):
      rtt_sample -= ack.delay
    UpdateRtt(rtt_sample)
  // Find all newly acked packets.
  for acked_packet_number in DetermineNewlyAckedPackets():
    OnPacketAcked(acked_packet_number)

  DetectLostPackets(ack.largest_acked_packet)
  SetLossDetectionAlarm()


UpdateRtt(rtt_sample):
  // Based on {{RFC6298}}.
  if (smoothed_rtt == 0):
    smoothed_rtt = rtt_sample
    rttvar = rtt_sample / 2
  else:
    rttvar = 3/4 * rttvar + 1/4 * (smoothed_rtt - rtt_sample)
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * rtt_sample
```

3.6. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAcked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAcked must be called once for each of these newly acked packets.

OnPacketAcked takes one parameter, acked_packet, which is the packet number of the newly acked packet, and returns a list of packet numbers that are detected as lost.

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet_number):
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  sent_packets.remove(acked_packet_number)
```

3.7. Setting the Loss Detection Alarm

QUIC loss detection uses a single alarm for all timer-based loss detection. The duration of the alarm is based on the alarm's mode, which is set in the packet and timer events further below. The function SetLossDetectionAlarm defined below shows how the single timer is set based on the alarm mode.

3.7.1. Handshake Packets

The initial flight has no prior RTT sample. A client SHOULD remember the previous RTT it observed when resumption is attempted and use that for an initial RTT value. If no previous RTT is available, the initial RTT defaults to 200ms. Once an RTT measurement is taken, it MUST replace initial_rtt.

Endpoints MUST retransmit handshake frames if not acknowledged within a time limit. This time limit will start as the largest of twice the rtt value and MinTLPTimeout. Each consecutive handshake retransmission doubles the time limit, until an acknowledgement is received.

Handshake frames may be cancelled by handshake state transitions. In particular, all non-protected frames SHOULD be no longer be transmitted once packet protection is available.

When stateless rejects are in use, the connection is considered immediately closed once a reject is sent, so no timer is set to retransmit the reject.

Version negotiation packets are always stateless, and MUST be sent once per per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0RTT packets.

3.7.2. Tail Loss Probe and Retransmission Timeout

Tail loss probes [I-D.dukkipati-tcpm-tcp-loss-probe] and retransmission timeouts[RFC6298] are an alarm based mechanism to recover from cases when there are outstanding retransmittable packets, but an acknowledgement has not been received in a timely manner.

3.7.3. Early Retransmit

Early retransmit [RFC5827] is implemented with a 1/4 RTT timer. It is part of QUIC's time based loss detection, but is always enabled, even when only packet reordering loss detection is enabled.

3.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
 SetLossDetectionAlarm():
    if (retransmittable packets are not outstanding):
      loss_detection_alarm.cancel();
      return

    if (handshake packets are outstanding):
      // Handshake retransmission alarm.
      if (smoothed_rtt == 0):
        alarm_duration = 2 * initial_rtt
      else:
        alarm_duration = 2 * smoothed_rtt
      alarm_duration = max(alarm_duration, kMinTLPTimeout)
      alarm_duration = alarm_duration << handshake_count
    else if (loss_time != 0):
      // Early retransmit timer or time loss detection.
      alarm_duration = loss_time - now
    else if (tlp_count < kMaxTLPs):
      // Tail Loss Probe
      if (retransmittable_packets_outstanding = 1):
        alarm_duration = 1.5 * smoothed_rtt + kDelayedAckTimeout
      else:
        alarm_duration = kMinTLPTimeout
      alarm_duration = max(alarm_duration, 2 * smoothed_rtt)
    else:
      // RTO alarm
      if (rto_count = 0):
        alarm_duration = smoothed_rtt + 4 * rttvar
        alarm_duration = max(alarm_duration, kMinRTOTimeout)
      else:
        alarm_duration = loss_detection_alarm.get_delay() << 1

    loss_detection_alarm.set(now + alarm_duration)
```

3.8. On Alarm Firing

QUIC uses one loss recovery alarm, which when set, can be in one of several modes. When the alarm fires, the mode determines the action to be performed.

Pseudocode for OnLossDetectionAlarm follows:

```
OnLossDetectionAlarm():
  if (handshake packets are outstanding):
    // Handshake retransmission alarm.
    RetransmitAllHandshakePackets();
    handshake_count++;
  // TODO: Clarify early retransmit and time loss.
  else if (loss_time != 0):
    // Early retransmit or Time Loss Detection
    DetectLostPackets(largest_acked_packet)
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe.
    if (HasNewDataToSend()):
      SendOnePacketOfNewData()
    else:
      RetransmitOldestPacket()
    tlp_count++
  else:
    // RTO.
    RetransmitOldestTwoPackets()
    rto_count++

  SetLossDetectionAlarm()
```

3.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time an ack is received. If the loss detection alarm fires and the loss_time is set, the previous largest acked packet is supplied.

3.9.1. Handshake Packets

The receiver MUST ignore unprotected packets that ack protected packets. The receiver MUST trust protected acks for unprotected packets, however. Aside from this, loss detection for handshake packets when an ack is processed is identical to other packets.

3.9.2. Pseudocode

DetectLostPackets takes one parameter, acked, which is the largest acked packet.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(largest_acked):
  loss_time = 0
  lost_packets = {}
  delay_until_lost = infinite;
  if (time_reordering_fraction != infinite):
    delay_until_lost =
      (1 + time_reordering_fraction) * max(latest_rtt, smoothed_rtt)
  else if (largest_acked.packet_number == largest_sent_packet):
    // Early retransmit alarm.
    delay_until_lost = 9/8 * max(latest_rtt, smoothed_rtt)
  foreach (unacked less than largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    packet_delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost):
      lost_packets.insert(unacked)
    else if (packet_delta > reordering_threshold)
      lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = delay_until_lost - time_since_sent

  // Inform the congestion controller of lost packets and
  // lets it decide whether to retransmit immediately.
  OnPacketsLost(lost_packets)
  foreach (packet in lost_packets)
    sent_packets.remove(packet.packet_number)
```

4. Congestion Control

(describe NewReno-style congestion control [RFC6582] for QUIC.) (describe appropriate byte counting.) (define recovery based on packet numbers.) (describe min_rtt based hystart.) (describe how QUIC's F-RTO [RFC5682] delays reducing CWND.) (describe PRR [RFC6937])

5. IANA Considerations

This document has no IANA actions. Yet.

6. References

6.1. Normative References

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

6.2. Informative References

[I-D.dukkipati-tcpm-tcp-loss-probe] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <http://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <http://www.rfc-editor.org/info/rfc5827>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <http://www.rfc-editor.org/info/rfc6298>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <http://www.rfc-editor.org/info/rfc6582>.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <http://www.rfc-editor.org/info/rfc6937>.

Appendix A. Acknowledgments

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-recovery-01

- Changes initial default RTT to 100ms
- Added time-based loss detection and fixes early retransmit
- Clarified loss recovery for handshake packets
- Fixed references and made TCP references informative

B.2. Since draft-ietf-quic-recovery-00:

- Improved description of constants and ACK behavior

B.3. Since draft-iyengar-quic-loss-recovery-01:

- Adopted as base for draft-ietf-quic-recovery.
- Updated authors/editors list.
- Added table of contents.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Ian Swett (editor)
Google

Email: ianswett@google.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd

March 13, 2017

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-02

Abstract

This document describes how Transport Layer Security (TLS) can be used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/tls .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction

QUIC [QUIC-TRANSPORT] provides a multiplexed transport. When used for HTTP [RFC7230] semantics [QUIC-HTTP] it provides several key advantages over HTTP/1.1 [RFC7230] or HTTP/2 [RFC7540] over TCP [RFC0793].

This document describes how QUIC can be secured using Transport Layer Security (TLS) version 1.3 [I-D.ietf-tls-tls13]. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how the standardized TLS 1.3 can act a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

2. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3.

TLS terminology is used when referring to parts of TLS. Though TLS assumes a continuous stream of octets, it divides that stream into _records_. Most relevant to QUIC are the records that contain TLS _handshake messages_, which are discrete messages that are used for key agreement, authentication and parameter negotiation. Ordinarily, TLS records can also contain _application data_, though in the QUIC usage there is no use of TLS application data.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS 1.3 connection [I-D.ietf-tls-tls13]; QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability and ordered delivery provided by QUIC streams.

This document defines how QUIC interacts with TLS. This includes a description of how TLS is used, how keying material is derived from TLS, and the application of that keying material to protect QUIC packets. Figure 1 shows the basic interactions between TLS and QUIC, with the QUIC packet protection being called out specially.

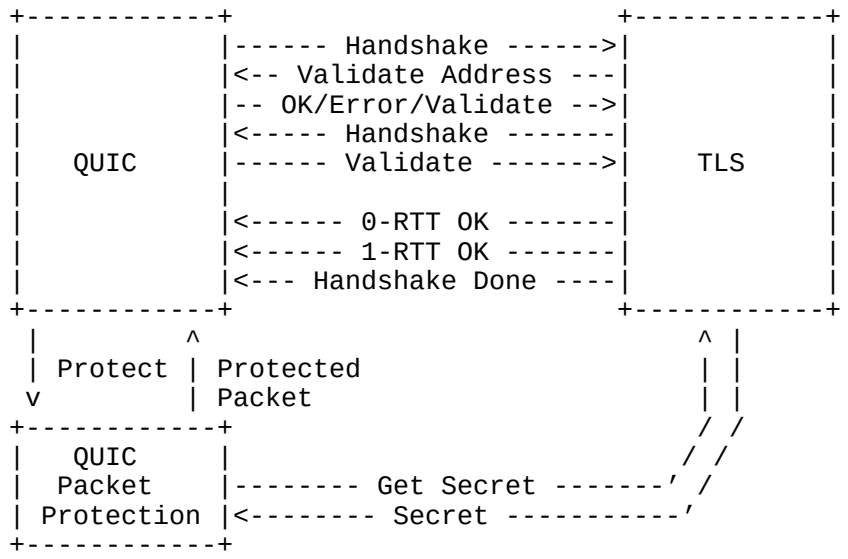


Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 1 and associated packets. Stream 1 is reserved for a TLS connection. This is a complete TLS connection as it would appear when layered over TCP; the only difference is that QUIC provides the reliability and ordering that would otherwise be provided by TCP.

At certain points during the TLS handshake, keying material is exported from the TLS connection for use by QUIC. This keying material is used to derive packet protection keys. Details on how and when keys are derived and used are included in Section 5.

This arrangement means that some TLS messages receive redundant protection from both the QUIC packet protection and the TLS record protection. These messages are limited in number; the TLS connection is rarely needed once the handshake completes.

3.1. TLS Overview

TLS provides two endpoints a way to establish a means of communication over an untrusted medium (that is, the Internet) that ensures that messages they exchange cannot be observed, modified, or forged.

TLS features can be separated into two basic functions: an authenticated key exchange and record protection. QUIC primarily

uses the authenticated key exchange provided by TLS but provides its own packet protection.

The TLS authenticated key exchange occurs between two entities: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman (DH) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the DH keys are destroyed.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 certificate-based authentication [RFC5280] for both server and client.

The TLS key exchange is resistent to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

3.2. TLS Handshake

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- A full, 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately after receiving the first handshake message from the client.

- A 0-RTT handshake in which the client uses information it has previously learned about the server to send immediately. This data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [I-D.ietf-tls-tls13] for more options and details.

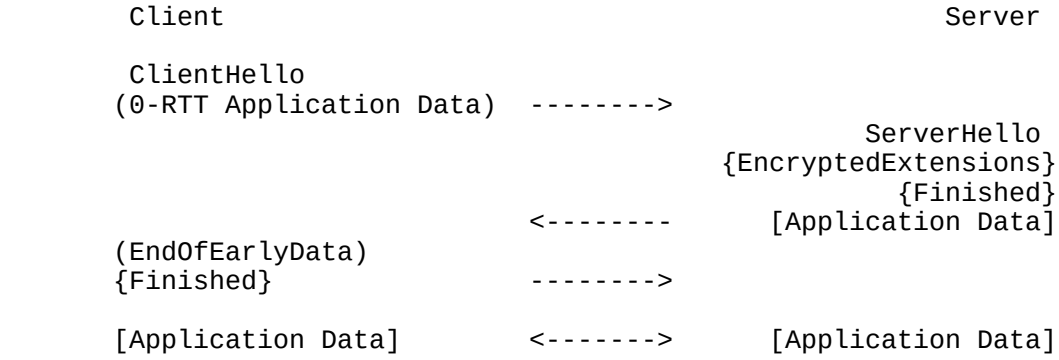


Figure 2: TLS Handshake with 0-RTT

This 0-RTT handshake is only possible if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

Two additional variations on this basic handshake exchange are relevant to this document:

- The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see [QUIC-TRANSPORT]).

- A pre-shared key mode can be used for subsequent handshakes to avoid public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

4. TLS Usage

QUIC reserves stream 1 for a TLS connection. Stream 1 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section-TBD), TLS is unmodified for this use. This means that TLS will apply confidentiality and integrity protection to its records. In particular, TLS record protection is what provides confidentiality protection for the TLS handshake messages sent by the server.

QUIC permits a client to send frames on streams starting from the first packet. The initial packet from a client contains a stream frame for stream 1 that contains the first TLS handshake messages

from the client. This allows the TLS handshake to start with the first packet that a client sends.

QUIC packets are protected using a scheme that is specific to QUIC, see Section 5. Keys are exported from the TLS connection when they become available using a TLS exporter (see Section 7.3.3 of [I-D.ietf-tls-tls13] and Section 5.2). After keys are exported from TLS, QUIC manages its own key schedule.

4.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 1 carry the TLS handshake. QUIC performs loss recovery [QUIC-RECOVERY] for this stream and ensures that TLS handshake messages are delivered in the correct order.

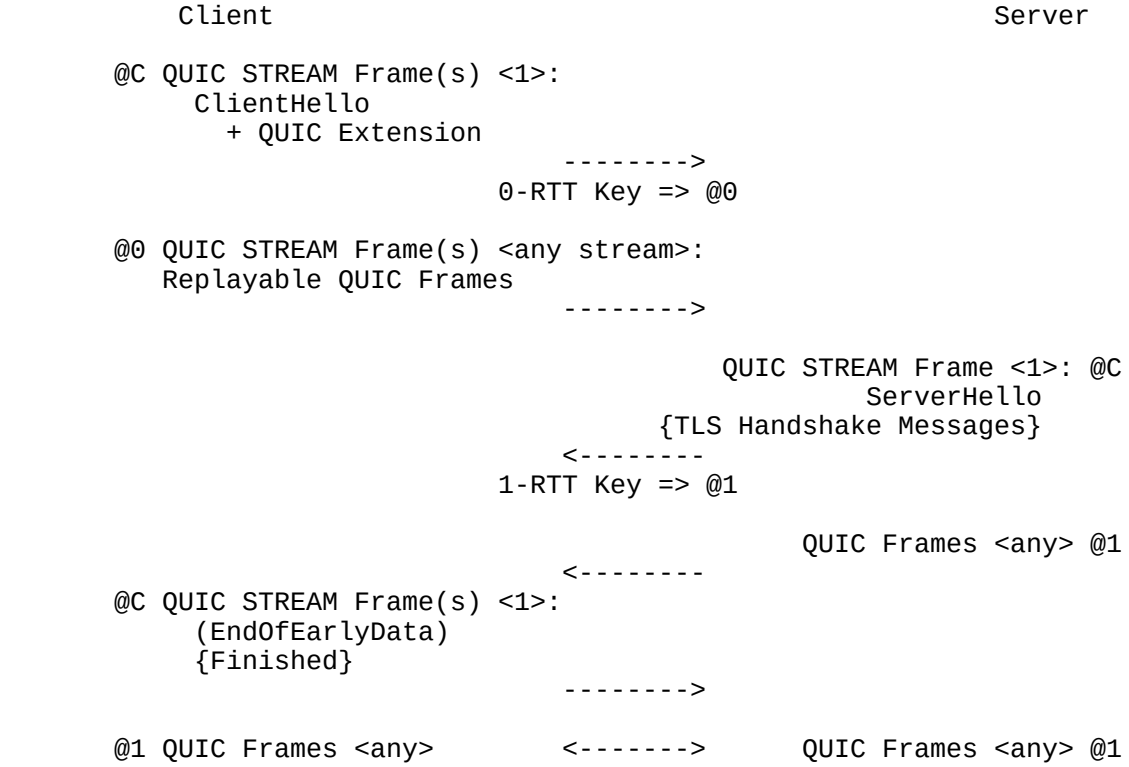


Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.
- "@" indicates the key phase that is currently used for protecting QUIC packets.
- "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.
- "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not attempted, then the client does not send packets protected by the 0-RTT key (@0). In that case, the only key transition on the client is from unprotected packets (@C) to 1-RTT protection (@1), which happens after it sends its final set of TLS handshake messages.

The server sends TLS handshake messages without protection (@C). The server transitions from no protection (@C) to full 1-RTT protection (@1) after it sends the last of its handshake messages.

Some TLS handshake messages are protected by the TLS handshake record protection. These keys are not exported from the TLS connection for use in QUIC. QUIC packets from the server are sent in the clear until the final transition to 1-RTT keys.

The client transitions from cleartext (@C) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to to 1-RTT keys (@1) after its second flight of TLS handshake messages. This creates the potential for unprotected packets to be received by a server in close proximity to packets that are protected with 1-RTT keys.

More information on key transitions is included in Section 6.1.

4.2. Interface to TLS

As shown in Figure 1, the interface from QUIC to TLS consists of four primary functions: Handshake, Source Address Validation, Key Ready Events, and Secret Export.

Additional functions might be needed to configure TLS.

4.2.1. Handshake Interface

In order to drive the handshake, TLS depends on being able to send and receive handshake messages on stream 1. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 9.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake octets from TLS. The client acquires handshake octets before sending its first packet.

A QUIC server starts the process by providing TLS with stream 1 octets.

Each time that an endpoint receives data on stream 1, it delivers the octets to TLS if it is able. Each time that TLS is provided with new data, new handshake octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives on stream 1. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client’s Finished message.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending a copy of the STREAM frame that carries the Finished message in multiple packets. This enables immediate server processing for those packets.

4.2.2. Source Address Validation

During the processing of the TLS ClientHello, TLS requests that the transport make a decision about whether to request source address validation from the client.

An initial TLS ClientHello that resumes a session includes an address validation token in the session ticket; this includes all attempts at 0-RTT. If the client does not attempt session resumption, no token will be present. While processing the initial ClientHello, TLS provides QUIC with any token that is present. In response, QUIC provides one of three responses:

- proceed with the connection,
- ask for client address validation, or
- abort the connection.

If QUIC requests source address validation, it also provides a new address validation token. TLS includes that along with any information it requires in the cookie extension of a TLS HelloRetryRequest message. In the other cases, the connection either proceeds or terminates with a handshake error.

The client echoes the cookie extension in a second ClientHello. A ClientHello that contains a valid cookie extension will be always be in response to a HelloRetryRequest. If address validation was requested by QUIC, then this will include an address validation token. TLS makes a second address validation request of QUIC, including the value extracted from the cookie extension. In response to this request, QUIC cannot ask for client address validation, it can only abort or permit the connection attempt to proceed.

QUIC can provide a new address validation token for use in session resumption at any time after the handshake is complete. Each time a new token is provided TLS generates a NewSessionTicket message, with the token included in the ticket.

See Section 7 for more details on client address validation.

4.2.3. Key Ready Events

TLS provides QUIC with signals when 0-RTT and 1-RTT keys are ready for use. These events are not asynchronous, they always occur immediately after TLS is provided with new handshake octets, or after TLS produces handshake octets.

When TLS completed its handshake, 1-RTT keys can be provided to QUIC. On both client and server, this occurs after sending the TLS Finished message.

This ordering means that there could be frames that carry TLS handshake messages ready to send at the same time that application data is available. An implementation MUST ensure that TLS handshake messages are always sent in cleartext packets. Separate packets are required for data that needs protection from 1-RTT keys.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal that 0-RTT keys are ready. On the server, after receiving handshake octets that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

1-RTT keys are used for packets in both directions. 0-RTT keys are only used to protect packets sent by the client.

4.2.4. Secret Export

Details how secrets are exported from TLS are included in Section 5.2.

4.2.5. TLS Interface Summary

Figure 4 summarizes the exchange between QUIC and TLS for both client and server.

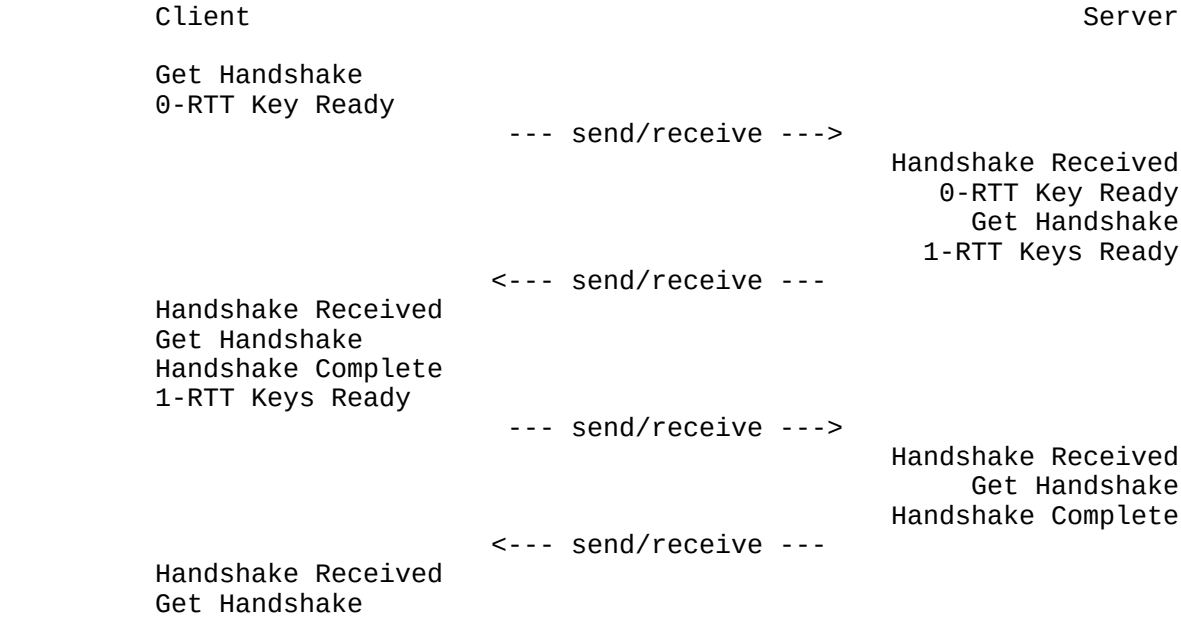


Figure 4: Interaction Summary between QUIC and TLS

4.3. TLS Version

This document describes how TLS 1.3 [I-D.ietf-tls-tls13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint MUST terminate the connection if a version of TLS older than 1.3 is negotiated.

4.4. ClientHello Size

QUIC requires that the initial handshake packet from a client fit within a single packet of at least 1280 octets. With framing and packet overheads this value could be reduced.

A TLS ClientHello can fit within this limit with ample space remaining. However, there are several variables that could cause this limit to be exceeded. Implementations are reminded that large

session tickets or HelloRetryRequest cookies, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, the size of the session tickets and HelloRetryRequest cookie extension can have an effect on a client's ability to connect. Choosing a small value increases the probability that these values can be successfully used by a client.

A TLS implementation does not need to enforce this size constraint. QUIC padding can be used to reach this size, meaning that a TLS server is unlikely to receive a large ClientHello message.

4.5. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client MUST authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server MUST NOT use post-handshake client authentication (see Section 4.6.2 of [I-D.ietf-tls-tls13]).

4.6. TLS Errors

Errors in the TLS connection SHOULD be signaled using TLS alerts on stream 1. A failure in the handshake MUST be treated as a QUIC connection error of type TLS_HANDSHAKE_FAILED. Once the handshake is complete, an error in the TLS connection that causes a TLS alert to be sent or received MUST be treated as a QUIC connection error of type TLS_FATAL_ALERT_GENERATED or TLS_FATAL_ALERT_RECEIVED respectively.

5. QUIC Packet Protection

QUIC packet protection provides authenticated encryption of packets. This provides confidentiality and integrity protection for the

content of packets (see Section 5.3). Packet protection uses keys that are exported from the TLS connection (see Section 5.2).

Different keys are used for QUIC packet protection and TLS record protection. Having separate QUIC and TLS record protection means that TLS records can be protected by two different keys. This redundancy is limited to only a few TLS records, and is maintained for the sake of simplicity.

5.1. Installing New Keys

As TLS reports the availability of keying material, the packet protection keys and initialization vectors (IVs) are updated (see Section 5.2). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any unprotected handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers MUST use the new keying material. The KEY_PHASE bit on these packets is inverted each time new keys are installed to signal the use of the new keys to the recipient (see Section 6 for details).

An endpoint retransmits stream data in a new packet. New packets have new packet numbers and use the latest packet protection keys. This simplifies key management when there are key updates (see Section 6.2).

5.2. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [I-D.ietf-tls-tls13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.3.3 of [I-D.ietf-tls-tls13]).

QUIC uses HKDF with the same hash function negotiated by TLS for key derivation. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the SHA-256 hash function is used.

5.2.1. 0-RTT Secret

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 8.2. 0-RTT keys are used after sending or receiving a ClientHello.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0-RTT Secret" and an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS. This uses the TLS early_exporter_secret. The QUIC 0-RTT secret is only used for protection of packets sent by the client.

```
client_0rtt_secret
    = TLS-Exporter("EXPORTER-QUIC 0-RTT Secret"
                   "", Hash.length)
```

5.2.2. 1-RTT Secrets

1-RTT keys are used by both client and server after the TLS handshake completes. There are two secrets used at any time: one is used to derive packet protection keys for packets sent by the client, the other for packet protection keys on packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1-RTT Secret"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1-RTT Secret". Both exporters use an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS.

```
client_pp_secret_0
    = TLS-Exporter("EXPORTER-QUIC client 1-RTT Secret"
                   "", Hash.length)
server_pp_secret_0
    = TLS-Exporter("EXPORTER-QUIC server 1-RTT Secret"
                   "", Hash.length)
```

These secrets are used to derive the initial client and server packet protection keys.

After a key update (see Section 6.2), these secrets are updated using the HKDF-Expand-Label function defined in Section 7.1 of [I-D.ietf-tls-tls13]. HKDF-Expand-Label uses the PRF hash function negotiated by TLS. The replacement secret is derived using the existing Secret, a Label of "QUIC client 1-RTT Secret" for the client and "QUIC server 1-RTT Secret" for the server, an empty HashValue, and the same output Length as the hash function selected by TLS for its PRF.

```
client_pp_secret_<N+1>
    = HKDF-Expand-Label(client_pp_secret_<N>,
                        "QUIC client 1-RTT Secret",
                        "", Hash.length)
server_pp_secret_<N+1>
    = HKDF-Expand-Label(server_pp_secret_<N>,
                        "QUIC server 1-RTT Secret",
                        "", Hash.length)
```

This allows for a succession of new secrets to be created as needed.

HKDF-Expand-Label uses HKDF-Expand [RFC5869] with a specially formatted info parameter. The info parameter that includes the output length (in this case, the size of the PRF hash output) encoded on two octets in network byte order, the length of the prefixed Label as a single octet, the value of the Label prefixed with "TLS 1.3, ", and a zero octet to indicate an empty HashValue. For example, the client packet protection secret uses an info parameter of:

```
info = (HashLen / 256) || (HashLen % 256) || 0x21 ||
       "TLS 1.3, QUIC client 1-RTT secret" || 0x00
```

5.2.3. Packet Protection Key and IV

The complete key expansion uses an identical process for key expansion as defined in Section 7.3 of [I-D.ietf-tls-tls13], using different values for the input secret. QUIC uses the AEAD function negotiated by TLS.

The packet protection key and IV used to protect the 0-RTT packets sent by a client use the QUIC 0-RTT secret. This uses the HKDF-Expand-Label with the PRF hash function negotiated by TLS.

The length of the output is determined by the requirements of the AEAD function selected by TLS. The key length is the AEAD key size. As defined in Section 5.3 of [I-D.ietf-tls-tls13], the IV length is the larger of 8 or N_MIN (see Section 4 of [RFC5116]).

```
client_0rtt_key = HKDF-Expand-Label(client_0rtt_secret,
                                    "key", "", key_length)
client_0rtt_iv = HKDF-Expand-Label(client_0rtt_secret,
                                   "iv", "", iv_length)
```

Similarly, the packet protection key and IV used to protect 1-RTT packets sent by both client and server use the current packet protection secret.

```
client_pp_key_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                      "key", "", key_length)
client_pp_iv_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                     "iv", "", iv_length)
server_pp_key_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                      "key", "", key_length)
server_pp_iv_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                     "iv", "", iv_length)
```

The client protects (or encrypts) packets with the client packet protection key and IV; the server protects packets with the server packet protection key.

The QUIC record protection initially starts without keying material. When the TLS state machine reports that the ClientHello has been sent, the 0-RTT keys can be generated and installed for writing. When the TLS state machine reports completion of the handshake, the 1-RTT keys can be generated and installed for writing.

5.3. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [RFC5116] function used for QUIC packet protection is AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

Regular QUIC packets are protected by an AEAD [RFC5116]. Version negotiation and public reset packets are not protected.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, K, for the AEAD is either the client packet protection key (client_pp_key_n) or the server packet protection key (server_pp_key_n), derived as defined in Section 5.2.

The nonce, N, for the AEAD is formed by combining either the packet protection IV (either client_pp_iv_n or server_pp_iv_n) with packet numbers. The 64 bits of the reconstructed QUIC packet number in network byte order is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the flags octet in the common header.

The input plaintext, P, for the AEAD is the contents of the QUIC frame following the packet number, as described in [QUIC-TRANSPORT].

The output ciphertext, C, of the AEAD is transmitted in place of P.

Prior to TLS providing keys, no record protection is performed and the plaintext, P, is transmitted unmodified.

5.4. Packet Numbers

QUIC has a single, contiguous packet number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending additional packets under the old key (causing new packets to be discarded).

QUIC does not assume a reliable transport and is required to handle attacks where packets are dropped in other ways. QUIC is therefore not affected by this form of truncation.

The QUIC packet number is not reset and it is not permitted to go higher than its maximum value of 2^64-1. This establishes a hard limit on the number of packets that can be sent.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV (see for example [AEBounds]). This might be lower than the packet number limit. An endpoint MUST initiate a key update (Section 6.2) prior to exceeding any limit set for the AEAD that is in use.

TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 1. This sequence number is not visible to QUIC.

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it MUST discard all packets with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - the next packet protection key (see Section 6.2). Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully MUST be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated

packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

6. Key Phases

As TLS reports the availability of 0-RTT and 1-RTT keys, new keying material can be exported from TLS and used for QUIC packet protection. At each transition during the handshake a new secret is exported from TLS and packet protection keys are derived from that secret.

Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled. The exception is the transition from 0-RTT keys to 1-RTT keys, where the presence of the version field and its associated bit is used (see Section 6.1.1).

Once the connection is fully enabled, the KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.2.

The KEY_PHASE bit is the third bit of the public flags (0x04).

Transitions between keys during the handshake are complicated by the need to ensure that TLS handshake messages are sent with the correct packet protection.

6.1. Packet Protection for the TLS Handshake

The initial exchange of packets are sent without protection. These packets are marked with a KEY_PHASE of 0.

TLS handshake messages MUST NOT be protected using QUIC packet protection. A KEY_PHASE of 0 is used for all of these packets, even during retransmission. The messages affected are all TLS handshake message up to the TLS Finished that is sent by each endpoint.

Any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. Packets containing these messages use the packet protection keys that are current at the time of sending (or retransmission).

Like the client, a server MUST send retransmissions of its unprotected handshake messages or acknowledgments for unprotected handshake messages sent by the client in unprotected packets (KEY_PHASE=0).

6.1.1. Initial Key Transitions

Once the TLS handshake is complete, keying material is exported from TLS and QUIC packet protection commences.

Packets protected with 1-RTT keys have a KEY_PHASE bit set to 1. These packets also have a VERSION bit set to 0.

If the client sends 0-RTT data, it marks packets protected with 0-RTT keys with a KEY_PHASE of 1 and a VERSION bit of 1. Setting the version bit means that all packets also include the version field. The client retains the VERSION bit, but reverts the KEY_PHASE bit for the packet that contains the TLS EndOfEarlyData and Finished messages.

The client clears the VERSION bit and sets the KEY_PHASE bit to 1 when it transitions to using 1-RTT keys.

Marking 0-RTT data with the both KEY_PHASE and VERSION bits ensures that the server is able to identify these packets as 0-RTT data in case packets containing TLS handshake message are lost or delayed. Including the version also ensures that the packet format is known to the server in this case.

Using both KEY_PHASE and VERSION also ensures that the server is able to distinguish between cleartext handshake packets (KEY_PHASE=0, VERSION=1), 0-RTT protected packets (KEY_PHASE=1, VERSION=1), and 1-RTT protected packets (KEY_PHASE=1, VERSION=0). Packets with all of these markings can arrive concurrently, and being able to identify each cleanly ensures that the correct packet protection keys can be selected and applied.

A server might choose to retain 0-RTT packets that arrive before a TLS ClientHello. The server can then use those packets once the ClientHello arrives. However, the potential for denial of service from buffering 0-RTT packets is significant. These packets cannot be authenticated and so might be employed by an attacker to exhaust server resources. Limiting the number of packets that are saved might be necessary.

The server transitions to using 1-RTT keys after sending its first flight of TLS handshake messages. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys and marked with a KEY_PHASE of 1.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

TLS handshake messages from both client and server are critical to the key exchange. The contents of these messages determines the keys used to protect later messages. If these handshake messages are included in packets that are protected with these keys, they will be indecipherable to the recipient.

Even though newer keys could be available when retranmitting, retransmissions of these handshake messages MUST be sent in unprotected packets (with a KEY_PHASE of 0). An endpoint MUST also generate ACK frames for these messages that are sent in unprotected packets.

A HelloRetryRequest handshake message might be used to reject an initial ClientHello. A HelloRetryRequest handshake message and any second ClientHello that is sent in response MUST also be sent without packet protection. This is natural, because no new keying material will be available when these messages need to be sent. Upon receipt of a HelloRetryRequest, a client SHOULD cease any transmission of 0-RTT data; 0-RTT data will only be discarded by any server that sends a HelloRetryRequest.

The KEY_PHASE and VERSION bits ensure that protected packets are clearly distinguished from unprotected packets. Loss or reordering might cause unprotected packets to arrive once 1-RTT keys are in use, unprotected packets are easily distinguished from 1-RTT packets.

Once 1-RTT keys are available to an endpoint, it no longer needs the TLS handshake messages that are carried in unprotected packets. However, a server might need to retransmit its TLS handshake messages in response to receiving an unprotected packet that contains ACK frames. A server MUST process ACK frames in unprotected packets until the TLS handshake is reported as complete, or it receives an ACK frame in a protected packet that acknowledges all of its handshake messages.

To limit the number of key phases that could be active, an endpoint MUST NOT initiate a key update while there are any unacknowledged handshake messages, see Section 6.2.

6.2. Key Update

Once the TLS handshake is complete, the KEY_PHASE bit allows for refreshes of keying material by either peer. Endpoints start using updated keys immediately without additional signaling; the change in the KEY_PHASE bit indicates that a new key is in use.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE. Note that when 0-RTT is attempted the value of the KEY_PHASE bit will be different on packets sent by either peer.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see Section 5.2) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

An endpoint doesn't need to send packets immediately when it detects that its peer has updated keys. The next packet that it sends will simply use the new keys. If an endpoint detects a second update before it has sent any packets with updated keys it indicates that its peer has updated keys twice without awaiting a reciprocal update. An endpoint MUST treat consecutive key updates as a fatal error and abort the connection.

An endpoint SHOULD retain old keys for a short period to allow it to decrypt packets with smaller packet numbers than the packet that triggered the key update. This allows an endpoint to consume packets that are reordered around the transition between keys. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with sequence numbers lower than the lowest sequence number used for the new key. An endpoint might discard keys if it determines that the length of the delay to affected packets is excessive.

This ensures that once the handshake is complete, packets with the same KEY_PHASE will have the same packet protection keys, unless there are multiple key updates in a short time frame succession and significant packet reordering.

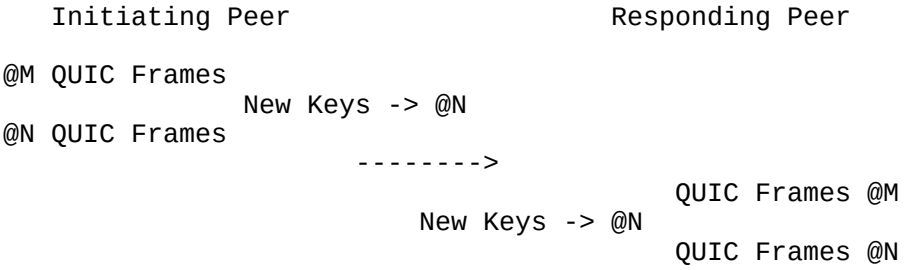


Figure 5: Key Update

As shown in Figure 3 and Figure 5, there is never a situation where there are more than two different sets of keying material that might be received by a peer. Once both sending and receiving keys have been updated,

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until all of its handshake messages have been acknowledged by the server.

A packet that triggers a key update could arrive after successfully processing a packet with a higher packet number. This is only possible if there is a key compromise and an attack, or if the peer is incorrectly reverting to use of old keys. Because the latter cannot be differentiated from an attack, an endpoint MUST immediately terminate the connection if it detects this condition.

7. Client Address Validation

Two tools are provided by TLS to enable validation of client source addresses at a server: the cookie in the HelloRetryRequest message, and the ticket in the NewSessionTicket message.

7.1. HelloRetryRequest Address Validation

The cookie extension in the TLS HelloRetryRequest message allows a server to perform source address validation during the handshake.

When QUIC requests address validation during the processing of the first ClientHello, the token it provides is included in the cookie extension of a HelloRetryRequest. As long as the cookie cannot be successfully guessed by a client, the server can be assured that the client received the HelloRetryRequest if it includes the value in a second ClientHello.

An initial ClientHello never includes a cookie extension. Thus, if a server constructs a cookie that contains all the information necessary to reconstruct state, it can discard local state after sending a HelloRetryRequest. Presence of a valid cookie in a ClientHello indicates that the ClientHello is a second attempt from the client.

An address validation token can be extracted from a second ClientHello and passed to the transport for further validation. If that validation fails, the server MUST fail the TLS handshake and send an illegal_parameter alert.

Combining address validation with the other uses of HelloRetryRequest ensures that there are fewer ways in which an additional round-trip can be added to the handshake. In particular, this makes it possible to combine a request for address validation with a request for a different client key share.

If TLS needs to send a HelloRetryRequest for other reasons, it needs to ensure that it can correctly identify the reason that the HelloRetryRequest was generated. During the processing of a second ClientHello, TLS does not need to consult the transport protocol regarding address validation if address validation was not requested originally. In such cases, the cookie extension could either be absent or it could indicate that an address validation token is not present.

7.2. NewSessionTicket Address Validation

The ticket in the TLS NewSessionTicket message allows a server to provide a client with a similar sort of token. When a client resumes a TLS connection - whether or not 0-RTT is attempted - it includes the ticket in the handshake message. As with the HelloRetryRequest cookie, the server includes the address validation token in the ticket. TLS provides the token it extracts from the session ticket to the transport when it asks whether source address validation is needed.

If both a HelloRetryRequest cookie and a session ticket are present in the ClientHello, only the token from the cookie is passed to the transport. The presence of a cookie indicates that this is a second ClientHello - the token from the session ticket will have been provided to the transport when it appeared in the first ClientHello.

A server can send a NewSessionTicket message at any time. This allows it to update the state - and the address validation token - that is included in the ticket. This might be done to refresh the ticket or token, or it might be generated in response to changes in

the state of the connection. QUIC can request that a NewSessionTicket be sent by providing a new address validation token.

A server that intends to support 0-RTT SHOULD provide an address validation token immediately after completing the TLS handshake.

7.3. Address Validation Token Integrity

TLS MUST provide integrity protection for address validation token unless the transport guarantees integrity protection by other means. For a NewSessionTicket that includes confidential information - such as the resumption secret - including the token under authenticated encryption ensures that the token gains both confidentiality and integrity protection without duplicating the overheads of that protection.

8. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 1 without packet protection. QUIC requires the use of several types of frame for managing loss detection and recovery during this phase. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion control.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server’s responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- discard and ignore them
- use them, but reset any state that is established once the handshake completes
- use them and authenticate them afterwards; failing the handshake if they can’t be authenticated
- save them and use them when they can be properly authenticated

- treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- Transport parameters are made usable and authenticated as part of the TLS handshake (see Section 9.2).

- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 8.1).

- Protected packets can either be discarded or saved and later used (see Section 8.3).

8.1. Unprotected Packets Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

8.1.1. STREAM Frames

"STREAM" frames for stream 1 are permitted. These carry the TLS handshake messages. Once 1-RTT keys are available, unprotected "STREAM" frames on stream 1 can be ignored.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

8.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an unprotected "ACK" frame to acknowledge data that was protected by 0-RTT or 1-RTT keys. An endpoint MUST ignore an unprotected "ACK" frame if it claims to acknowledge data that was sent in a protected packet. Such an acknowledgement can only serve

as a denial of service, since an endpoint that can read protected data is always able to send protected data.

ISSUE: What about 0-RTT data? Should we allow acknowledgment of 0-RTT with unprotected frames? If we don't, then 0-RTT data will be unacknowledged until the handshake completes. This isn't a problem if the handshake completes without loss, but it could mean that 0-RTT stalls when a handshake packet disappears for any reason.

An endpoint SHOULD use data from unprotected or 0-RTT-protected "ACK" frames only during the initial handshake and while they have insufficient information from 1-RTT-protected "ACK" frames. Once sufficient information has been obtained from protected messages, information obtained from less reliable sources can be discarded.

8.1.3. WINDOW_UPDATE Frames

"WINDOW_UPDATE" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 1, the initial flow control window is sufficiently large to allow the TLS handshake to complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 1 is exempt from the connection-level flow control window.

8.1.4. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching sequence number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. From that point onward, unprotected messages can be safely dropped.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

ISSUE: This would not be an issue if QUIC had a randomized starting sequence number. If we choose to randomize, we fix this problem and reduce the denial of service exposure to on-path attackers. The only possible problem is in authenticating the initial value, so that peers can be sure that they haven't missed an initial message.

In addition to causing valid packets to be dropped, an attacker can generate packets with an intent of causing the recipient to expend processing resources. See Section 10.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation MUST reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts that is received prior to the end of the handshake MUST be treated as a fatal error.

8.2. Use of 0-RTT Keys

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

8.3. Receiving Out-of-Order Protected Frames

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client.

Packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete. A server MUST NOT use 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key -

the pre-shared key binder (see Section 4.2.8 of [I-D.ietf-tls-tls13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

Receiving and verifying the TLS Finished message is critical in ensuring the integrity of the TLS handshake. A server MUST NOT use protected packets from the client prior to verifying the client Finished message if its response depends on client authentication.

9. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

9.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade during the handshake, though it means that such a downgrade causes a handshake failure.

TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] to select an application protocol. The application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected.

If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

9.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different format for this struct.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {
   quic_transport_parameters(26), (65535)
} ExtensionType;
```

The "extension_data" field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use. The quic_transport_parameters extension carries a TransportParameters when the version of QUIC defined in [QUIC-TRANSPORT] is used.

9.3. Priming 0-RTT

QUIC uses TLS without modification. Therefore, it is possible to use a pre-shared key that was obtained in a TLS connection over TCP to enable 0-RTT in QUIC. Similarly, QUIC can provide a pre-shared key that can be used to enable 0-RTT in TCP.

All the restrictions on the use of 0-RTT apply, with the exception of the ALPN label, which MUST only change to a label that is explicitly designated as being compatible. The client indicates which ALPN label it has chosen by placing that ALPN label first in the ALPN extension.

The certificate that the server uses MUST be considered valid for both connections, which will use different protocol stacks and could use different port numbers. For instance, HTTP/1.1 and HTTP/2 operate over TLS and TCP, whereas QUIC operates over UDP.

Source address validation is not completely portable between different protocol stacks. Even if the source IP address remains constant, the port number is likely to be different. Packet reflection attacks are still possible in this situation, though the set of hosts that can initiate these attacks is greatly reduced. A server might choose to avoid source address validation for such a connection, or allow an increase to the amount of data that it sends toward the client without source validation.

10. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

10.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

QUIC requires that the packet containing a ClientHello be padded to the size of the maximum transmission unit (MTU). A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of this size. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to significantly exceed the size of the packet containing the ClientHello.

10.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to generate unnecessary work. Once the TLS handshake is complete, endpoints SHOULD NOT send TLS application data records unless it is to hide the length of QUIC records. QUIC packet protection does not include any allowance for padding; padded TLS application data records can be used to mask the length of QUIC frames.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

11. Error codes

The portion of the QUIC error code space allocated for the crypto handshake is 0xC0000000-0xFFFFFFFF. The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0xC000001C): The TLS handshake failed.

TLS_FATAL_ALERT_GENERATED (0xC000001D): A TLS fatal alert was sent, causing the TLS connection to end prematurely.

TLS_FATAL_ALERT_RECEIVED (0xC000001E): A TLS fatal alert was received, causing the TLS connection to end prematurely.

12. IANA Considerations

This document has no IANA actions. Yet.

13. References

13.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

13.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC".

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <http://www.rfc-editor.org/info/rfc7924>.

Appendix A. Contributors

Ryan Hamilton was originally an author of this specification.

Appendix B. Acknowledgments

This document has benefited from input from Dragana Damjanovic, Christian Huitema, Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and many others.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-01:

- Use TLS alerts to signal TLS errors (#272, #374)
- Require ClientHello to fit in a single packet (#338)
- The second client handshake flight is now sent in the clear (#262, #337)
- The QUIC header is included as AEAD Associated Data (#226, #243, #302)
- Add interface necessary for client address validation (#275)
- Define peer authentication (#140)
- Require at least TLS 1.3 (#138)
- Define transport parameters as a TLS extension (#122)
- Define handling for protected packets before the handshake completes (#39)
- Decouple QUIC version and ALPN (#12)

C.2. Since draft-ietf-quic-tls-00:

- Changed bit used to signal key phase.
- Updated key phase markings during the handshake.

- Added TLS interface requirements section.
- Moved to use of TLS exporters for key derivation.
- Moved TLS error code definitions into this document.

C.3. Since draft-thomson-quic-tls-01:

- Adopted as base for draft-ietf-quic-tls.
- Updated authors/editors list.
- Added status note.

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
Google
M. Thomson, Ed.
Mozilla

March 13, 2017

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-02

Abstract

This document defines the core of the QUIC transport protocol. This document describes connection establishment, packet format, multiplexing and reliability. Accompanying documents describe the cryptographic handshake and loss detection.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/transport .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Table of Contents

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow it to be a general-purpose transport for multiple applications.

QUIC implements techniques learned from experience with TCP, SCTP and other transport protocols. Using UDP as the substrate, QUIC seeks to be compatible with legacy clients and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes.

This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability.

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS 1.3 for key negotiation [QUIC-TLS].

2. Conventions and Definitions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.

Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.

Connection ID: The identifier for a QUIC connection.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

2.1. Notational Conventions

Packet and frame diagrams use the format described in [RFC2360] Section 3.1, with the following additional conventions:

[x] Indicates that x is optional

{x} Indicates that x is encrypted

x (A) Indicates that x is A bits long

x (A/B/C) ... Indicates that x is one of A, B, or C bits long

x (*) ... Indicates that x is variable-length

3. A QUIC Overview

This section briefly describes QUIC’s key mechanisms and benefits. Key strengths of QUIC include:

- Low-latency connection establishment
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Rich signaling for congestion control and loss recovery
- Stream and connection flow control
- Connection migration and resilience to NAT rebinding
- Version negotiation

3.1. Low-Latency Connection Establishment

QUIC relies on a combined cryptographic and transport handshake for setting up a secure transport connection. QUIC connections are expected to commonly use 0-RTT handshakes, meaning that for most QUIC connections, data can be sent immediately following the client handshake packet, without waiting for a reply from the server. QUIC provides a dedicated stream (Stream ID 1) to be used for performing the cryptographic handshake and QUIC options negotiation. The format of the QUIC options and parameters used during negotiation are described in this document, but the handshake protocol that runs on Stream ID 1 is described in the accompanying cryptographic handshake draft [QUIC-TLS].

3.2. Stream Multiplexing

When application messages are transported over TCP, independent application messages can suffer from head-of-line blocking. When an application multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the application streams that are encapsulated in subsequent segments. QUIC ensures that lost packets carrying data for an individual stream only impact that specific stream. Data received on other streams can continue to be reassembled and delivered to the application.

3.3. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet carries a new packet number, including those carrying retransmitted data. This obviates the need for a separate mechanism to distinguish acknowledgments for retransmissions from those for original transmissions, avoiding TCP's retransmission ambiguity problem. QUIC acknowledgments also explicitly encode the delay between the receipt of a packet and its acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise network roundtrip-time (RTT) calculation. QUIC's ACK frames support up to 256 ACK blocks, so QUIC is more resilient to reordering than TCP with SACK support, as well as able to keep more bytes on the wire when there is reordering or loss.

3.4. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control mechanisms. At a high level, a QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent,

received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. In addition to this stream-level flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to all streams on a connection. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

3.5. Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and are not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are mechanisms used by middleboxes to improve TCP performance, others are active attacks. Even "performance-enhancing" middleboxes that routinely interpose on the transport state machine end up limiting the evolvability of the transport protocol, as has been observed in the design of MPTCP [RFC6824] and in its subsequent deployability issues.

Generally, QUIC packets are always authenticated and the payload is typically fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. Some early handshake packets, such as the Version Negotiation packet, are not encrypted, but information sent in these unencrypted handshake packets is later verified as part of cryptographic processing.

PUBLIC_RESET packets that reset a connection are currently not authenticated.

3.6. Connection Migration and Resilience to NAT Rebinding

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC’s consistent connection ID allows connections to survive changes to the client’s IP and port, such as those caused by NAT rebindings or by the client changing network connectivity to a new address. QUIC provides automatic cryptographic verification of a rebound client, since the client continues to use the same session key for encrypting and decrypting packets. The consistent connection ID can be used to allow migration of the connection to a new server IP address as well, since the Connection ID remains consistent across changes in the client’s and the server’s network addresses.

3.7. Version Negotiation

QUIC version negotiation allows for multiple versions of the protocol to be deployed and used concurrently. Version negotiation is described in Section 7.1.

4. Versions

QUIC versions are identified using a 32-bit value.

The version 0x00000000 is reserved to represent an invalid version. This version of the specification is identified by the number 0x00000001.

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all octets is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC.

Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-quic-transport-13 would be identified as 0xff00000D.

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the github wiki [4].

5. Packet Types and Formats

We first describe QUIC’s packet types and their formats, since some are referenced in subsequent mechanisms.

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. When discussing individual bits of fields, the least significant bit is referred to as bit 0. Hexadecimal notation is used for describing the value of fields.

Any QUIC packet has either a long or a short header, as indicated by the Header Form bit. Long headers are expected to be used early in the connection before version negotiation and establishment of 1-RTT keys, and for public resets. Short headers are minimal version-specific headers, which can be used after version negotiation and 1-RTT keys are established.

5.1. Long Header

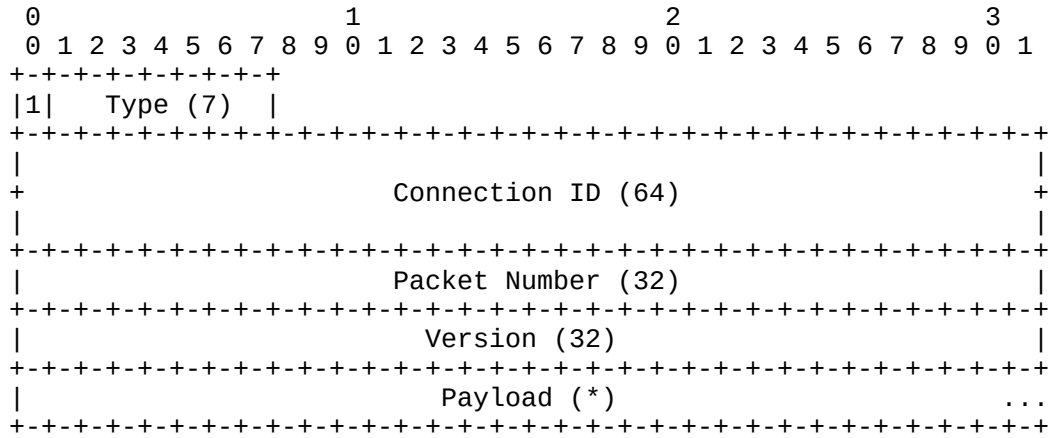


Figure 1: Long Header Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender SHOULD switch to sending short-form headers. While inefficient, long headers MAY be used for packets encrypted with 1-RTT keys. The long form allows for special packets, such as the Version Negotiation and the Public Reset packets to be represented in this uniform fixed-length packet format. A long header contains the following fields:

Header Form: The most significant bit (0x80) of the first octet is set to 1 for long headers and 0 for short headers.

Long Packet Type: The remaining seven bits of first octet of a long packet is the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 1.

Connection ID: Octets 1 through 8 contain the connection ID. Section 5.7 describes the use of this field in more detail.

Packet Number: Octets 9 to 12 contain the packet number. {{packet-numbers} describes the use of packet numbers.

Version: Octets 13 to 16 contain the selected protocol version. This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

Payload: Octets from 17 onwards (the rest of QUIC packet) are the payload of the packet.

The following packet types are defined:

Type	Name	Section
01	Version Negotiation	Section 5.3
02	Client Cleartext	Section 5.4
03	Non-Final Server Cleartext	Section 5.4
04	Final Server Cleartext	Section 5.4
05	0-RTT Encrypted	Section 5.5
06	1-RTT Encrypted (key phase 0)	Section 5.5
07	1-RTT Encrypted (key phase 1)	Section 5.5
08	Public Reset	Section 5.6

Table 1: Long Header Packet Types

The header form, packet type, connection ID, packet number and version fields of a long header packet are version-independent. The types of packets defined in Table 1 are version-specific. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

(TODO: Should the list of packet types be version-independent?)

The interpretation of the fields and the payload are specific to a version and packet type. Type-specific semantics for this version

are described in Section 5.3, Section 5.6, Section 5.4, and Section 5.5.

5.2. Short Header

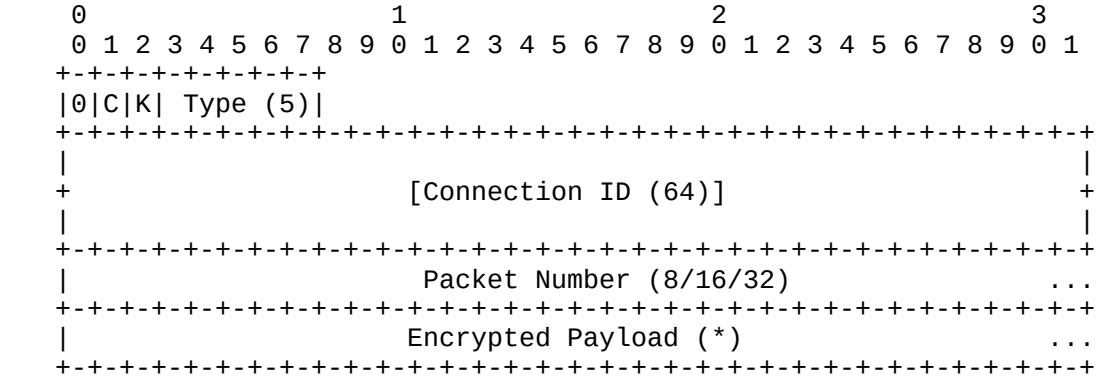


Figure 2: Short Header Format

The short header can be used after the version and 1-RTT keys are negotiated. This header form has the following fields:

Header Form: The most significant bit (0x80) of the first octet of a packet is the header form. This bit is set to 0 for the short header.

Connection ID Flag: The second bit (0x40) of the first octet indicates whether the Connection ID field is present. If set to 1, then the Connection ID field is present; if set to 0, the Connection ID field is omitted.

Key Phase Bit: The third bit (0x20) of the first octet indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

Short Packet Type: The remaining 5 bits of the first octet include one of 32 packet types. Table 2 lists the types that are defined for short packets.

Connection ID: If the Connection ID Flag is set, a connection ID occupies octets 1 through 8 of the packet. See Section 5.7 for more details.

Packet Number: The length of the packet number field depends on the packet type. This field can be 1, 2 or 4 octets long depending on the short packet type.

Encrypted Payload: Packets with a short header always include a 1-RTT protected payload.

The packet type in a short header currently determines only the size of the packet number field. Additional types can be used to signal the presence of other fields.

Type	Packet Number Size
01	1 octet
02	2 octets
03	4 octets

Table 2: Short Header Packet Types

The header form, connection ID flag and connection ID of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

5.3. Version Negotiation Packet

A Version Negotiation packet is sent only by servers and is a response to a client packet of an unsupported version. It uses a long header and contains:

- Octet 0: 0x81
- Octets 1-8: Connection ID (echoed)
- Octets 9-12: Packet Number (echoed)
- Octets 13-16: Version (echoed)
- Octets 17+: Payload

The payload of the Version Negotiation packet is a list of 32-bit versions which the server supports, as shown below.

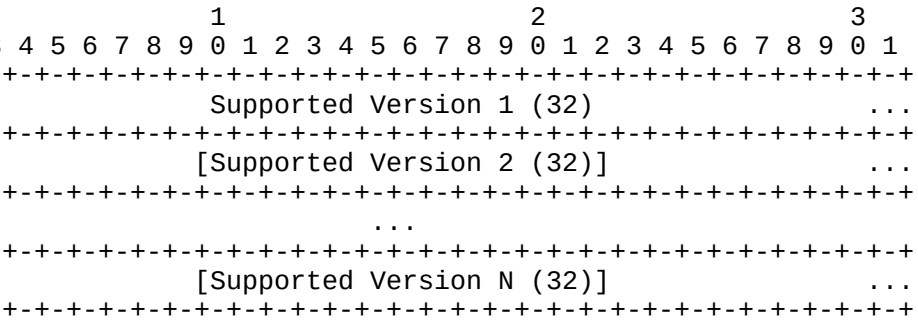


Figure 3: Version Negotiation Packet

See Section 7.1 for a description of the version negotiation process.

5.4. Cleartext Packets

Cleartext packets are sent during the handshake prior to key negotiation. A Client Cleartext packet contains:

- Octet 0: 0x82
- Octets 1-8: Connection ID (initial)
- Octets 9-12: Packet number
- Octets 13-16: Version
- Octets 17+: Payload

Non-Final Server Cleartext packets contain:

- Octet 0: 0x83
- Octets 1-8: Connection ID (echoed)
- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Payload

Final Server Cleartext packets contains:

- Octet 0: 0x84
- Octets 1-8: Connection ID (final)

- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Payload

The client MUST choose a random 64-bit value and use it as the initial Connection ID in all packets until the server replies with the final Connection ID. The server echoes the client's Connection ID in Non-Final Server Cleartext packets. The first Final Server Cleartext and all subsequent packets MUST use the final Connection ID, as described in Section 5.7.

The payload of a Cleartext packet consists of a sequence of frames, as described in Section 6.

(TODO: Add hash before frames.)

5.5. Encrypted Packets

Packets encrypted with either 0-RTT or 1-RTT keys may be sent with long headers. Different packet types explicitly indicate the encryption level for ease of decryption. These packets contain:

- Octet 0: 0x85, 0x86 or 0x87
- Octets 1-8: Connection ID (initial or final)
- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Encrypted Payload

A first octet of 0x85 indicates a 0-RTT packet. After the 1-RTT keys are established, key phases are used by the QUIC packet protection to identify the correct packet protection keys. The initial key phase is 0. See [QUIC-TLS] for more details.

The encrypted payload is both authenticated and encrypted using packet protection keys. [QUIC-TLS] describes packet protection in detail. After decryption, the plaintext consists of a sequence of frames, as described in Section 6.

5.6. Public Reset Packet

A Public Reset packet is only sent by servers and is used to abruptly terminate communications. Public Reset is provided as an option of last resort for a server that does not have access to the state of a connection. This is intended for use by a server that has lost state (for example, through a crash or outage). A server that wishes to communicate a fatal connection error MUST use a CONNECTION_CLOSE frame if it has sufficient state to do so.

A Public Reset packet contains:

- Octet 0: 0x88
- Octets 1-8: Echoed data (octets 1-8 of received packet)
- Octets 9-12: Echoed data (octets 9-12 of received packet)
- Octets 13-16: Version
- Octets 17+: Public Reset Proof

For a client that sends a connection ID on every packet, the Connection ID field is simply an echo of the initial Connection ID, and the Packet Number field includes an echo of the client's packet number (and, depending on the client's packet number length, 0, 2, or 3 additional octets from the client's packet).

A Public Reset packet sent by a server indicates that it does not have the state necessary to continue with a connection. In this case, the server will include the fields that prove that it originally participated in the connection (see Section 5.6.1 for details).

Upon receipt of a Public Reset packet that contains a valid proof, a client MUST tear down state associated with the connection. The client MUST then cease sending packets on the connection and SHOULD discard any subsequent packets that arrive. A Public Reset that does not contain a valid proof MUST be ignored.

5.6.1. Public Reset Proof

TODO: Details to be added.

5.7. Connection ID

QUIC connections are identified by their 64-bit Connection ID. All long headers contain a Connection ID. Short headers indicate the presence of a Connection ID using the CONNECTION_ID flag. When present, the Connection ID is in the same location in all packet headers, making it straightforward for middleboxes, such as load balancers, to locate and use it.

When a connection is initiated, the client MUST choose a random value and use it as the initial Connection ID until the final value is available. The initial Connection ID is a suggestion to the server. The server echoes this value in all packets until the handshake is successful (see [QUIC-TLS]). On a successful handshake, the server MUST select the final Connection ID for the connection and use it in Final Server Cleartext packets. This final Connection ID MAY be the one proposed by the client or MAY be a new server-selected value. All subsequent packets from the server MUST contain this value. On handshake completion, the client MUST switch to using the final Connection ID for all subsequent packets.

Thus, all Client Cleartext packets, 0-RTT Encrypted packets, and Non-Final Server Cleartext packets MUST use the client's randomly-generated initial Connection ID. Final Server Cleartext packets, 1-RTT Encrypted packets, and all short-header packets MUST use the final Connection ID.

5.8. Packet Numbers

The packet number is a 64-bit unsigned number and is used as part of a cryptographic nonce for packet encryption. Each endpoint maintains a separate packet number for sending and receiving. The packet number for sending MUST increase by at least one after sending any packet.

A QUIC endpoint MUST NOT reuse a packet number within the same connection (that is, under the same cryptographic keys). If the packet number for sending reaches 2^64 - 1, the sender MUST close the connection by sending a CONNECTION_CLOSE frame with the error code QUIC_SEQUENCE_NUMBER_LIMIT_REACHED (connection termination is described in Section 7.6.)

To reduce the number of bits required to represent the packet number over the wire, only the least significant bits of the packet number are transmitted over the wire, up to 32 bits. The actual packet number for each packet is reconstructed at the receiver based on the largest packet number received on a successfully authenticated packet.

A packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 16-bit value of 0x1f94 will be decoded as 0xaa831f94.

The sender MUST use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint MAY use a larger packet number size to safeguard against such reordering.

As a result, the size of the packet number encoding is at least one more than the base 2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet.

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b4264 requires a 16-bit or larger packet number encoding; whereas a 32-bit packet number is needed to send a packet with a number of 0x6bc107.

5.8.1. Initial Packet Number

The initial value for packet number MUST be a 31-bit random number. That is, the value is selected from an uniform random distribution between 0 and 2^31-1. [RFC4086] provides guidance on the generation of random values.

The first set of packets sent by an endpoint MUST include the low 32-bits of the packet number. Once any packet has been acknowledged, subsequent packets can use a shorter packet number encoding.

5.9. Handling Packets from Different Versions

Between different versions the following things are guaranteed to remain constant:

- the location of the header form flag,
- the location of the Connection ID flag in short headers,
- the location and size of the Connection ID field in both header forms,
- the location and size of the Version field in long headers, and

o the location and size of the Packet Number field in long headers.

Implementations MUST assume that an unsupported version uses an unknown packet format. All other fields MUST be ignored when processing a packet that contains an unsupported version.

6. Frames and Frame Types

The payload of cleartext packets and the plaintext after decryption of encrypted payloads consists of a sequence of frames, as shown in Figure 4.

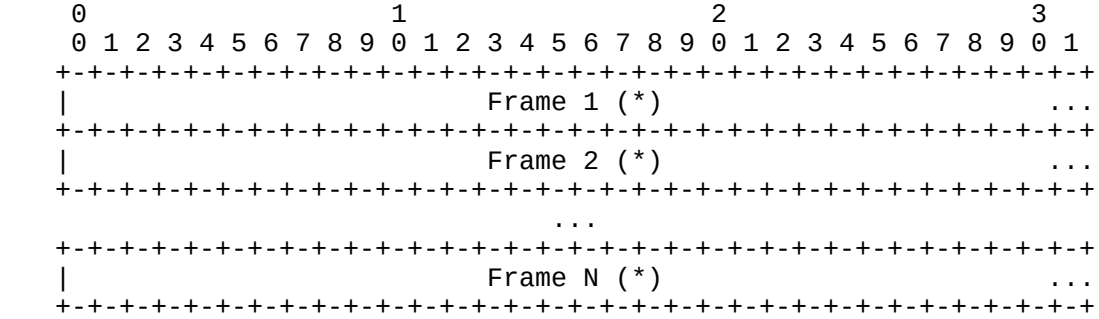


Figure 4: Contents of Encrypted Payload

Encrypted payloads MUST contain at least one frame, and MAY contain multiple frames and multiple frame types.

Frames MUST fit within a single QUIC packet and MUST NOT span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by additional type-dependent fields:

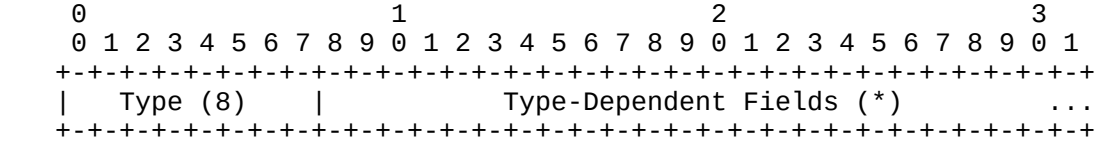


Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM and ACK frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in the document.

Type-field value	Frame type	Definition
0x00	PADDING	Section 8.6
0x01	RST_STREAM	Section 8.5
0x02	CONNECTION_CLOSE	Section 8.8
0x03	GOAWAY	Section 8.9
0x04	WINDOW_UPDATE	Section 8.3
0x05	BLOCKED	Section 8.4
0x07	PING	Section 8.7
0x40 - 0x7f	ACK	Section 8.2
0x80 - 0xff	STREAM	Section 8.1

Table 3: Frame Types

7. Life of a Connection

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment intertwines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 7.2. Once established, a connection may migrate to a different IP or port at either endpoint, due to NAT rebinding or mobility, as described in Section 7.5. Finally a connection may be terminated by either endpoint, as described in Section 7.6.

7.1. Version Negotiation

QUIC's connection establishment begins with version negotiation, since all communication between the endpoints, including packet and frame formats, relies on the two endpoints agreeing on a version.

A QUIC connection begins with a client sending a handshake packet. The details of the handshake mechanisms are described in Section 7.2, but all of the initial packets sent from the client to the server MUST use the long header format and MUST specify the version of the protocol being used.

When the server receives a packet from a client with the long header format, it compares the client's version to the versions it supports.

If the version selected by the client is not acceptable to the server, the server discards the incoming packet and responds with a Version Negotiation packet (Section 5.3). This includes a list of versions that the server will accept. A server MUST send a Version Negotiation packet for every packet that it receives with an unacceptable version.

If the packet contains a version that is acceptable to the server, the server proceeds with the handshake (Section 7.2). This commits the server to the version that the client selected.

When the client receives a Version Negotiation packet from the server, it should select an acceptable protocol version. If the server lists an acceptable version, the client selects that version and reattempts to create a connection using that version. Though the contents of a packet might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers and MUST include the new negotiated protocol version.

The client MUST use the long header format and include its selected version on all packets until it has 1-RTT keys and it has received a packet from the server which is not a Version Negotiation packet.

A client MUST NOT change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it MUST ignore Version Negotiation packets on the same connection.

Version negotiation uses unprotected data. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 7.3.4).

7.1.1. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 4) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. However, when the server generates a Version Negotiation packet, it cannot randomly generate a reserved version number. This is because

the server is required to include the same value in its transport parameters (see Section 7.3.4). To avoid the selected version number changing during connection establishment, the reserved version SHOULD be generated as a function of values that will be available to the server when later generating its handshake packets.

A pseudorandom function that takes client address information (IP and port) and the client selected version as input would ensure that there is sufficient variability in the values that a server uses.

A client MAY send a packet using a reserved version number. This can be used to solicit a list of supported versions from a server.

7.2. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC allocates stream 1 for the cryptographic handshake. This version of QUIC uses TLS 1.3 [QUIC-TLS].

QUIC provides this stream with reliable, ordered delivery of data. In return, the cryptographic handshake provides QUIC with:

- authenticated key exchange, where
  - a server is always authenticated,
  - a client is optionally authenticated,
  - every connection produces distinct and unrelated keys,
  - keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and
  - 1-RTT keys have forward secrecy
- authenticated values for the transport parameters of the peer (see Section 7.3)
- authenticated confirmation of version negotiation (see Section 7.3.4)
- authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)
- for the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client (see Section 7.4)

The initial cryptographic handshake message MUST be sent in a single packet. Any second attempt that is triggered by address validation MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets. Reassembling messages requires that a server maintain state prior to establishing a connection, exposing the server to a denial of service risk.

The first client packet of the cryptographic handshake protocol MUST fit within a 1280 octet QUIC packet. This includes overheads that reduce the space available to the cryptographic handshake protocol.

Details of how TLS is integrated with QUIC is provided in more detail in [QUIC-TLS].

7.3. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. These declarations are made unilaterally by each endpoint. Endpoints are required to comply with the restrictions implied by these parameters; the description of each parameter includes rules for its handling.

The format of the transport parameters is the TransportParameters struct from Figure 6. This is described using the presentation language from Section 3 of [I-D.ietf-tls-tls13].

```
uint32 QuicVersion;

enum {
   stream_fc_offset(0),
   connection_fc_offset(1),
   concurrent_streams(2),
   idle_timeout(3),
   truncate_connection_id(4),
   (65535)
} TransportParameterId;

struct {
   TransportParameterId parameter;
   opaque value<0..2^16-1>;
} TransportParameter;

struct {
   select (Handshake.msg_type) {
      case client_hello:
         QuicVersion negotiated_version;
         QuicVersion initial_version;

      case encrypted_extensions:
         QuicVersion supported_versions<2..2^8-4>;
   };
   TransportParameter parameters<30..2^16-1>;
} TransportParameters;
```

Figure 6: Definition of TransportParameters

The "extension_data" field of the quic_transport_parameters extension defined in [QUIC-TLS] contains a TransportParameters value. TLS encoding rules are therefore used to encode the transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST be validated (see Section 7.3.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 7.3.1.

7.3.1. Transport Parameter Definitions

An endpoint MUST include the following parameters in its encoded TransportParameters:

stream_fc_offset (0x0000): The initial stream level flow control offset parameter is encoded as an unsigned 32-bit integer in units of octets. The sender of this parameter indicates that the flow control offset for all stream data sent toward it is this value.

connection_fc_offset (0x0001): The connection level flow control offset parameter contains the initial connection flow control window encoded as an unsigned 32-bit integer in units of 1024 octets. That is, the value here is multiplied by 1024 to determine the actual flow control offset. The sender of this parameter sets the byte offset for connection level flow control to this value. This is equivalent to sending a WINDOW_UPDATE (Section 8.3) for the connection immediately after completing the handshake.

concurrent_streams (0x0002): The maximum number of concurrent streams parameter is encoded as an unsigned 32-bit integer.

idle_timeout (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. The maximum value is 600 seconds (10 minutes).

An endpoint MAY use the following transport parameters:

truncate_connection_id (0x0004): The truncated connection identifier parameter indicates that packets sent to the peer can omit the connection ID. This can be used by an endpoint where the 5-tuple is sufficient to identify a connection. This parameter is zero length. Omitting the parameter indicates that the endpoint relies on the connection ID being present in every packet.

7.3.2. Values of Transport Parameters for 0-RTT

Transport parameters from the server SHOULD be remembered by the client for use with 0-RTT data. A client that doesn’t remember values from a previous connection can instead assume the following values: stream_fc_offset (65535), connection_fc_offset (65535), concurrent_streams (10), idle_timeout (600), truncate_connection_id (absent).

If assumed values change as a result of completing the handshake, the client is expected to respect the new values. This introduces some

potential problems, particularly with respect to transport parameters that establish limits:

- A client might exceed a newly declared connection or stream flow control limit with 0-RTT data. If this occurs, the client ceases transmission as though the flow control limit was reached. Once WINDOW_UPDATE frames indicating an increase to the affected flow control offsets is received, the client can recommence sending.

- Similarly, a client might exceed the concurrent stream limit declared by the server. A client MUST reset any streams that exceed this limit. A server SHOULD reset any streams it cannot handle with a code that allows the client to retry any application action bound to those streams.

A server MAY close a connection if remembered or assumed 0-RTT transport parameters cannot be supported, using an error code that is appropriate to the specific condition. For example, a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA might be used to indicate that exceeding flow control limits caused the error. A client that has a connection closed due to an error condition SHOULD NOT attempt 0-RTT when attempting to create a new connection.

7.3.3. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter.

The definition of a transport parameter SHOULD include a default value that a client can use when establishing a new connection. If no default is specified, the value can be assumed to be absent when attempting 0-RTT.

New transport parameters can be registered according to the rules in Section 14.1.

7.3.4. Version Negotiation Validation

The transport parameters include three fields that encode version information. These retroactively authenticate the version negotiation (see Section 7.1) that is performed prior to the cryptographic handshake.

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see

Section 7.3). As a result, modification of version negotiation packets by an attacker can be detected.

The client includes two fields in the transport parameters:

- The negotiated_version is the version that was finally selected for use. This MUST be identical to the value that is on the packet that carries the ClientHello. A server that receives a negotiated_version that does not match the version of QUIC that is in use MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error code.

- The initial_version is the version that the client initially attempted to use. If the server did not send a version negotiation packet Section 5.3, this will be identical to the negotiated_version.

A server that processes all packets in a stateful fashion can remember how version negotiation was performed and validate the initial_version value.

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial and negotiated versions are the same, a stateless server can accept the value.

If the initial version is different from the negotiated_version, a stateless server MUST check that it would have sent a version negotiation packet if it had received a packet with the indicated initial_version. If a server would have accepted the version included in the initial_version and the value differs from the value of negotiated_version, the server MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error.

The server includes a list of versions that it would send in any version negotiation packet (Section 5.3) in supported_versions. This value is set even if it did not send a version negotiation packet.

The client can validate that the negotiated_version is included in the supported_versions list and - if version negotiation was performed - that it would have selected the negotiated version. A client MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error code if the negotiated_version value is not included in the supported_versions list. A client MUST terminate with a QUIC_VERSION_NEGOTIATION_MISMATCH error code if version negotiation occurred but it would have selected a different version based on the value of the supported_versions list.

7.4. Proof of Source Address Ownership

Transport protocols commonly spend a round trip checking that a client owns the transport address (IP and port) that it claims. Verifying that a client can receive packets sent to its claimed transport address protects against spoofing of this information by malicious clients.

This technique is used primarily to avoid QUIC from being used for traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

Several methods are used in QUIC to mitigate this attack. Firstly, the initial handshake packet from a client is padded to at least 1280 octets. This allows a server to send a similar amount of data without risking causing an amplication attack toward an unproven remote address.

A server eventually confirms that a client has received its messages when the cryptographic handshake successfully completes. This might be insufficient, either because the server wishes to avoid the computational cost of completing the handshake, or it might be that the size of the packets that are sent during the handshake is too large. This is especially important for 0-RTT, where the server might wish to provide application data traffic - such as a response to a request - in response to the data carried in the early data from the client.

To send additional data prior to completing the cryptographic handshake, the server then needs to validate that the client owns the address that it claims.

Source address validation is therefore performed during the establishment of a connection. TLS provides the tools that support the feature, but basic validation is performed by the core transport protocol.

7.4.1. Client Address Validation Procedure

QUIC uses token-based address validation. Any time the server wishes to validate a client address, it provides the client with a token. As long as the token cannot be easily guessed (see Section 7.4.3), if the client is able to return that token, it proves to the server that it received the token.

During the processing of the cryptographic handshake messages from a client, TLS will request that QUIC make a decision about whether to proceed based on the information it has. TLS will provide QUIC with any token that was provided by the client. For an initial packet, QUIC can decide to abort the connection, allow it to proceed, or request address validation.

If QUIC decides to request address validation, it provides the cryptographic handshake with a token. The contents of this token are consumed by the server that generates the token, so there is no need for a single well-defined format. A token could include information about the claimed client address (IP and port), a timestamp, and any other supplementary information the server will need to validate the token in the future.

The cryptographic handshake is responsible for enacting validation by sending the address validation token to the client. A legitimate client will include a copy of the token when it attempts to continue the handshake. The cryptographic handshake extracts the token then asks QUIC a second time whether the token is acceptable. In response, QUIC can either abort the connection or permit it to proceed.

A connection MAY be accepted without address validation - or with only limited validation - but a server SHOULD limit the data it sends toward an unvalidated address. Successful completion of the cryptographic handshake implicitly provides proof that the client has received packets from the server.

7.4.2. Address Validation on Session Resumption

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

A different type of token is needed when resuming. Unlike the token that is created during a handshake, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

This token can be provided to the cryptographic handshake immediately after establishing a connection. QUIC might also generate an updated token if significant time passes or the client address changes for

any reason (see Section 7.5). The cryptographic handshake is responsible for providing the client with the token. In TLS the token is included in the ticket that is used for resumption and 0-RTT, which is carried in a NewSessionTicket message.

7.4.3. Address Validation Token Integrity

An address validation token MUST be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

In TLS the address validation token is often bundled with the information that TLS requires, such as the resumption secret. In this case, adding integrity protection can be delegated to the cryptographic handshake protocol, avoiding redundant protection. If integrity protection is delegated to the cryptographic handshake, an integrity failure will result in immediate cryptographic handshake failure. If integrity protection is performed by QUIC, QUIC MUST abort the connection if the integrity check fails with a QUIC_ADDRESS_VALIDATION_FAILURE error code.

7.5. Connection Migration

QUIC connections are identified by their 64-bit Connection ID. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by client or server migrating to a new network. QUIC also provides automatic cryptographic verification of a client which has changed its IP address because the client continues to use the same session key for encrypting and decrypting packets.

DISCUSS: Simultaneous migration. Is this reasonable?

TODO: Perhaps move mitigation techniques from Security Considerations here.

7.6. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. A QUIC connection, once established, can be terminated in one of three ways:

1. Explicit Shutdown: An endpoint sends a CONNECTION_CLOSE frame to initiate a connection termination. An endpoint may send a GOAWAY frame to the peer prior to a CONNECTION_CLOSE to indicate that the connection will soon be terminated. A GOAWAY frame signals to the peer that any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

2. Implicit Shutdown: The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a CONNECTION_CLOSE frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.

3. Abrupt Shutdown: An endpoint may send a Public Reset packet at any time during the connection to abruptly terminate an active connection. A Public Reset packet SHOULD only be used as a final recourse. Commonly, a public reset is expected to be sent when a packet on an established connection is received by an endpoint that is unable decrypt the packet. For instance, if a server reboots mid-connection and loses any cryptographic state associated with open connections, and then receives a packet on an open connection, it should send a Public Reset packet in return. (TODO: articulate rules around when a public reset should be sent.)

TODO: Connections that are terminated are added to a TIME_WAIT list at the server, so as to absorb any straggler packets in the network. Discuss TIME_WAIT list.

8. Frame Types and Formats

As described in Section 6, Regular packets contain one or more frames. We now describe the various QUIC frame types that can be present in a Regular packet. The use of these frames and various frame header bits are described in subsequent sections.

8.1. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. The type byte for a STREAM frame contains embedded flags, and is formatted as "1FDOOOSS". These bits are parsed as follows:

- The leftmost bit must be set to 1, indicating that this is a STREAM frame.
- "F" is the FIN bit, which is used for stream termination.
- The "D" bit indicates whether a Data Length field is present in the STREAM header. When set to 0, this field indicates that the Stream Data field extends to the end of the packet. When set to 1, this field indicates that Data Length field contains the length (in bytes) of the Stream Data field. The option to omit the length should only be used when the packet is a "full-sized" packet, to avoid the risk of corruption via padding.
- The "OOO" bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
- The "SS" bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits. (DISCUSS: Consider making this 8, 16, 32, 64.)

A STREAM frame is shown below.

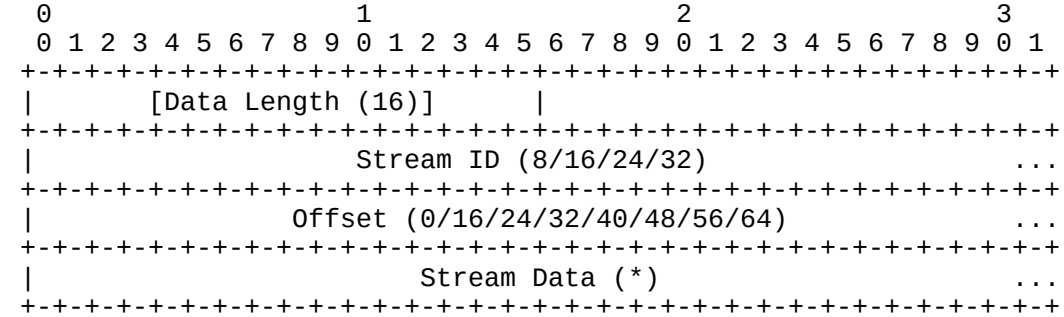


Figure 7: STREAM Frame Format

The STREAM frame contains the following fields:

Data Length: An optional 16-bit unsigned number specifying the length of the Stream Data field in this STREAM frame. This field is present when the "D" bit is set to 1.

Stream ID: A variable-sized unsigned ID unique to this stream.

Offset: A variable-sized unsigned number specifying the byte offset in the stream for the data in this STREAM frame. The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^64.

Stream Data: The bytes from the designated stream to be delivered.

A STREAM frame MUST have either non-zero data length or the FIN bit set.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet MAY bundle STREAM frames from multiple streams.

Implementation note: One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is bundled into a single QUIC packet, loss of that packet blocks all those streams from making progress. An implementation is therefore advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

8.2. ACK Frame

Receivers send ACK frames to inform senders which packets they have received and processed, as well as which packets are considered missing. The ACK frame contains between 1 and 256 ACK blocks. ACK blocks are ranges of acknowledged packets.

To limit ACK blocks to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. Once an ACK frame has been acknowledged, the packets it acknowledges SHOULD not be acknowledged again. To handle cases where the receiver is only sending ACK frames, and hence will not receive acknowledgments for its packets, it MAY send a PING frame at most once per RTT to explicitly request acknowledgment.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data.

Unlike TCP SACKs, QUIC ACK blocks are cumulative and therefore irrevocable. Once a packet has been acknowledged, even if it does not appear in a future ACK frame, it is assumed to be acknowledged.

QUIC ACK frames contain a timestamp section with up to 255 timestamps. Timestamps enable better congestion control, but are not required for correct loss recovery, and old timestamps are less valuable, so it is not guaranteed every timestamp will be received by the sender. A receiver SHOULD send a timestamp exactly once for each received packet containing retransmittable frames. A receiver MAY send timestamps for non-retransmittable packets.

A sender MAY intentionally skip packet numbers to introduce entropy into the connection, to avoid opportunistic acknowledgement attacks. The sender MUST close the connection if an unsent packet number is acknowledged. The format of the ACK frame is efficient at expressing blocks of missing packets; skipping packet numbers between 1 and 255 effectively provides up to 8 bits of efficient entropy on demand, which should be adequate protection against most opportunistic acknowledgement attacks.

The type byte for a ACK frame contains embedded flags, and is formatted as "01NULLMM". These bits are parsed as follows:

- The first two bits must be set to 01 indicating that this is an ACK frame.
- The "N" bit indicates whether the frame has more than 1 range of acknowledged packets (i.e., whether the ACK Block Section contains a Num Blocks field).
- The "U" bit is unused and MUST be set to zero.
- The two "LL" bits encode the length of the Largest Acknowledged field as 1, 2, 4, or 6 bytes long.
- The two "MM" bits encode the length of the ACK Block Length fields as 1, 2, 4, or 6 bytes long.

An ACK frame is shown below.

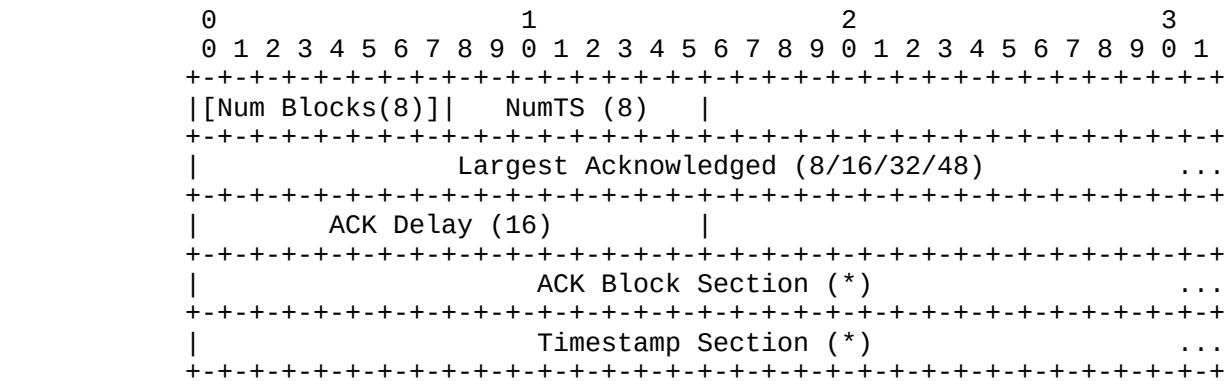


Figure 8: ACK Frame Format

The fields in the ACK frame are as follows:

Num Blocks (opt): An optional 8-bit unsigned value specifying the number of additional ACK blocks (besides the required First ACK Block) in this ACK frame. Only present if the 'N' flag bit is 1.

Num Timestamps: An unsigned 8-bit number specifying the total number of <packet number, timestamp> pairs in the Timestamp Section.

Largest Acknowledged: A variable-sized unsigned value representing the largest packet number the peer is acknowledging in this packet (typically the largest that the peer has seen thus far.)

ACK Delay: The time from when the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent.

ACK Block Section: Contains one or more blocks of packet numbers which have been successfully received, see Section 8.2.1.

Timestamp Section: Contains zero or more timestamps reporting transit delay of received packets. See Section 8.2.2.

8.2.1. ACK Block Section

The ACK Block Section contains between one and 256 blocks of packet numbers which have been successfully received. If the Num Blocks field is absent, only the First ACK Block length is present in this section. Otherwise, the Num Blocks field indicates how many additional blocks follow the First ACK Block Length field.

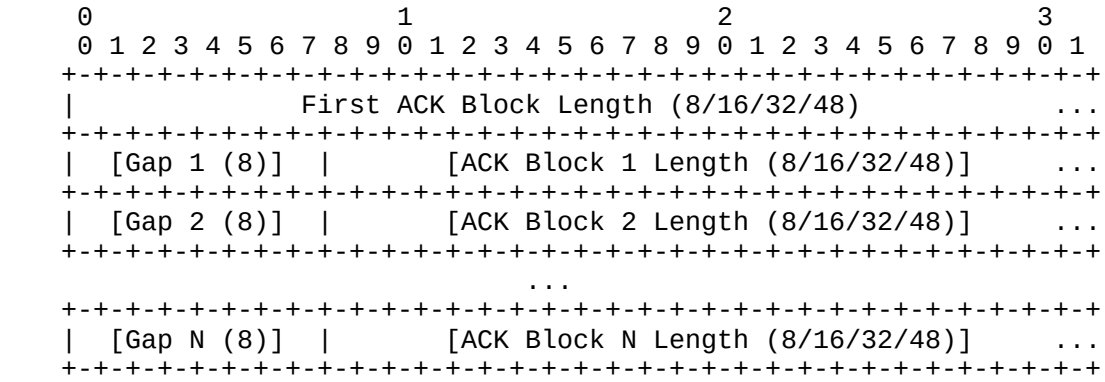


Figure 9: ACK Block Section

The fields in the ACK Block Section are:

First ACK Block Length: An unsigned packet number delta that indicates the number of contiguous additional packets being acknowledged starting at the Largest Acknowledged.

Gap To Next Block (opt, repeated): An unsigned number specifying the number of contiguous missing packets from the end of the previous ACK block to the start of the next. Repeated "Num Blocks" times.

ACK Block Length (opt, repeated): An unsigned packet number delta that indicates the number of contiguous packets being acknowledged starting after the end of the previous gap. Repeated "Num Blocks" times.

8.2.2. Timestamp Section

The Timestamp Section contains between zero and 255 measurements of packet receive times relative to the beginning of the connection.

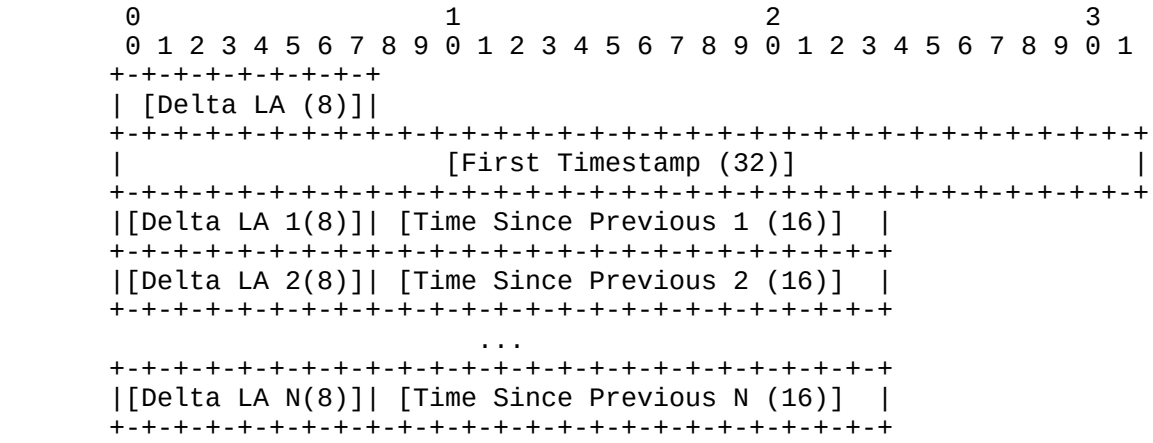


Figure 10: Timestamp Section

The fields in the Timestamp Section are:

Delta Largest Acknowledged (opt): An optional 8-bit unsigned packet number delta specifying the delta between the largest acknowledged and the first packet whose timestamp is being reported. In other words, this first packet number may be computed as (Largest Acknowledged - Delta Largest Acknowledged.)

First Timestamp (opt): An optional 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection to the arrival of the packet indicated by Delta Largest Acknowledged.

Delta Largest Acked 1..N (opt, repeated): This field has the same semantics and format as "Delta Largest Acknowledged". Repeated "Num Timestamps - 1" times.

Time Since Previous Timestamp 1..N(opt, repeated): An optional 16-bit unsigned value specifying time delta from the previous reported timestamp. It is encoded in the same format as the ACK Delay. Repeated "Num Timestamps - 1" times.

The timestamp section lists packet receipt timestamps ordered by timestamp.

8.2.2.1. Time Format

DISCUSS_AND_REPLACE: Perhaps make this format simpler.

The time format used in the ACK frame above is a 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying time in microseconds. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.

8.2.3. ACK Frames and Packet Protection

ACK frames that acknowledge protected packets MUST be carried in a packet that has an equivalent or greater level of packet protection.

Packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

A packet that is not protected and claims to acknowledge a packet number that was sent with packet protection is not valid. An unprotected packet that carries acknowledgments for protected packets MUST be discarded in its entirety.

Packets that a client sends with 0-RTT packet protection MUST be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

Unprotected packets, such as those that carry the initial cryptographic handshake messages, MAY be acknowledged in unprotected packets. Unprotected packets are vulnerable to falsification or modification. Unprotected packets can be acknowledged along with protected packets in a protected packet.

An endpoint SHOULD acknowledge packets containing cryptographic handshake messages in the next unprotected packet that it sends, unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected packets

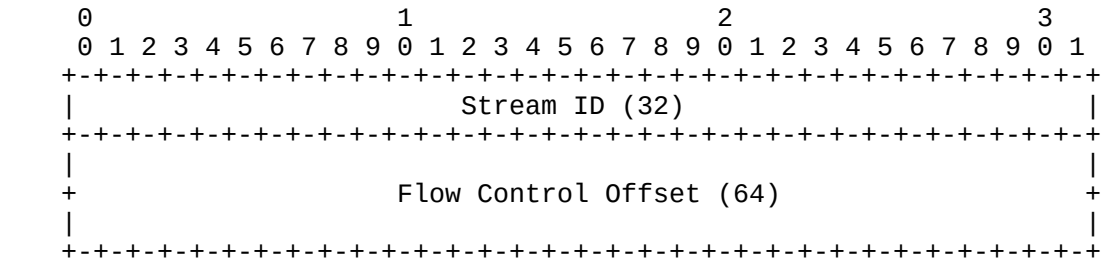
that complete the cryptographic handshake in a protected packet, because its peer is guaranteed to have access to 1-RTT packet protection keys.

For instance, a server acknowledges a TLS ClientHello in the packet that carries the TLS ServerHello; similarly, a client can acknowledge a TLS HelloRetryRequest in the packet containing a second TLS ClientHello. The complete set of server handshake messages (TLS ServerHello through to Finished) might be acknowledged by a client in protected packets, because it is certain that the server is able to decipher the packet.

8.3. WINDOW_UPDATE Frame

The WINDOW_UPDATE frame (type=0x04) informs the peer of an increase in an endpoint's flow control receive window for either a single stream, or the entire connection as a whole.

The frame is as follows:



The fields in the WINDOW_UPDATE frame are as follows:

Stream ID: ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.

Flow Control Offset: A 64-bit unsigned integer indicating the flow control offset for the given stream (for a stream ID other than 0) or the entire connection.

The flow control offset is expressed in units of octets for individual streams (for stream identifiers other than 0).

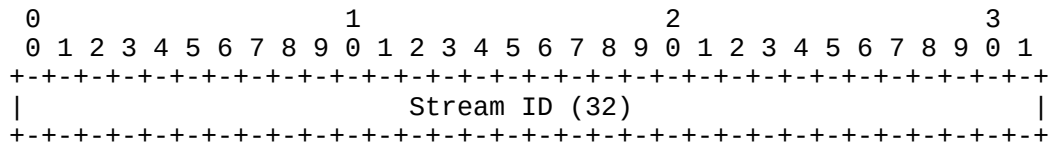
The connection-level flow control offset is expressed in units of 1024 octets (for a stream identifier of 0). That is, the connection-level flow control offset is determined by multiplying the encoded value by 1024.

An endpoint accounts for the maximum offset of data that is sent or received on a stream. Loss or reordering can mean that the maximum offset is greater than the total size of data received on a stream. Similarly, receiving STREAM frames might not increase the maximum offset on a stream. A STREAM frame with a FIN bit set or RST_STREAM causes the final offset for a stream to be fixed.

The maximum data offset on a stream MUST NOT exceed the stream flow control offset advertised by the receiver. The sum of the maximum data offsets of all streams (including closed streams) MUST NOT exceed the connection flow control offset advertised by the receiver. An endpoint MUST terminate a connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error if it receives more data than the largest flow control offset that it has sent, unless this is a result of a change in the initial offsets (see Section 7.3.2).

8.4. BLOCKED Frame

A sender sends a BLOCKED frame (type=0x05) when it is ready to send data (and has data to send), but is currently flow control blocked. BLOCKED frames are purely informational frames, but extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:

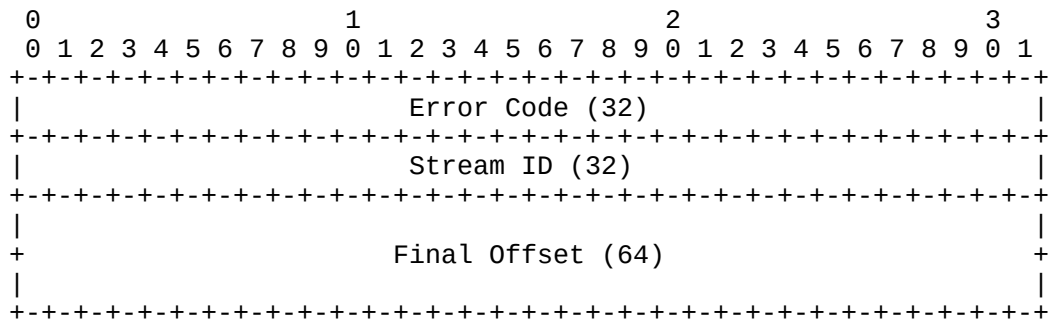


The BLOCKED frame contains a single field:

Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked.

8.5. RST_STREAM Frame

An endpoint may use a RST_STREAM frame (type=0x01) to abruptly terminate a stream. The frame is as follows:



The fields are:

Error code: A 32-bit error code which indicates why the stream is being closed.

Stream ID: The 32-bit Stream ID of the stream being terminated.

Final offset: A 64-bit unsigned integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

8.6. PADDING Frame

The PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

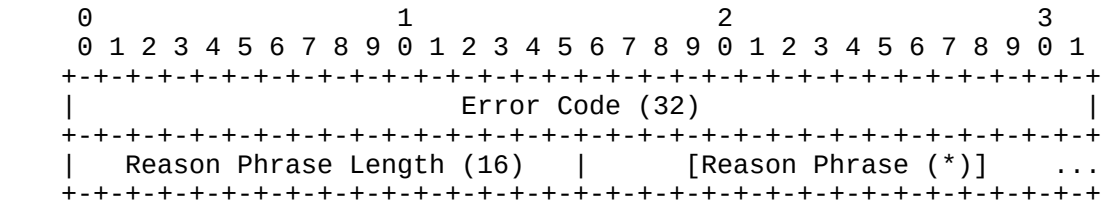
8.7. PING frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields. The receiver of a PING frame simply needs to acknowledge the packet containing this frame. The PING frame SHOULD be used to keep a connection alive when a stream is open. The default is to send a PING frame after 15 seconds of quiescence. A PING frame has no additional fields.

8.8. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. If there are open streams that haven’t been explicitly closed, they are implicitly closed when

the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:



The fields of a CONNECTION_CLOSE frame are as follows:

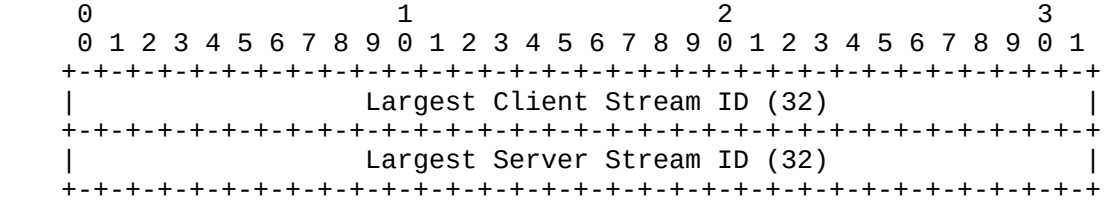
Error Code: A 32-bit error code which indicates the reason for closing this connection.

Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the Error Code.

Reason Phrase: An optional human-readable explanation for why the connection was closed.

8.9. GOAWAY Frame

An endpoint uses a GOAWAY frame (type=0x03) to initiate a graceful shutdown of a connection. The endpoints will continue to use any active streams, but the sender of the GOAWAY will not initiate or accept any additional streams beyond those indicated. The GOAWAY frame is as follows:



The fields of a GOAWAY frame are:

Largest Client Stream ID: The highest-numbered, client-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, client-initiated streams (that is, odd-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

Largest Server Stream ID: The highest-numbered, server-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, server-initiated streams (that is, even-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

A GOAWAY frame indicates that any application layer actions on streams with higher numbers than those indicated can be safely retried because no data was exchanged. An endpoint MUST set the value of the Largest Client or Server Stream ID to be at least as high as the highest-numbered stream on which it either sent data or received and delivered data to the application protocol that uses QUIC.

An endpoint MAY choose a larger stream identifier if it wishes to allow for a number of streams to be created. This is especially valuable for peer-initiated streams where packets creating new streams could be in transit; using a larger stream number allows those streams to complete.

In addition to initiating a graceful shutdown of a connection, GOAWAY MAY be sent immediately prior to sending a CONNECTION_CLOSE frame that is sent as a result of detecting a fatal error. Higher-numbered streams than those indicated in the GOAWAY frame can then be retried.

9. Packetization and Reliability

The Path Maximum Transmission Unit (PTMU) is the maximum size of the entire IP header, UDP header, and UDP payload. The UDP payload includes the QUIC public header, encrypted payload, and any authentication fields.

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery ([RFC4821]) and MAY use PMTU Discovery ([RFC1191], [RFC1981]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints SHOULD NOT send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a UDP payload length of 1232 octets for IPv6 and 1252 octets for IPv4.

QUIC endpoints that implement any kind of PMTU discovery SHOULD maintain an estimate for each combination of local and remote IP addresses (as each pairing could have a different maximum MTU in the path).

QUIC depends on the network path supporting a MTU of at least 1280 octets. This is the IPv6 minimum and therefore also supported by most modern IPv4 networks. An endpoint MUST NOT reduce their MTU below this number, even if it receives signals that indicate a smaller limit might exist.

Clients MUST ensure that the first packet in a connection, and any retransmissions of those octets, has a total size (including IP and UDP headers) of at least 1280 bytes. This might require inclusion of PADDING frames. It is RECOMMENDED that a packet be padded to exactly 1280 octets unless the client has a reasonable assurance that the PMTU is larger. Sending a packet of this size ensures that the network path supports an MTU of this size and helps mitigate amplification attacks caused by server responses toward an unverified client address.

Servers MUST reject the first plaintext packet received from a client if it its total size is less than 1280 octets, to mitigate amplification attacks.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it MUST immediately cease sending QUIC packets between those IP addresses. This may result in abrupt termination of the connection if all pairs are affected. In this case, an endpoint SHOULD send a Public Reset packet to indicate the failure. The application SHOULD attempt to use TLS over TCP instead.

A sender bundles one or more frames in a Regular QUIC packet (see Section 6).

A sender SHOULD minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender MAY wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation may use heuristics about expected application sending behavior to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Regular QUIC packets are "containers" of frames; a packet is never retransmitted whole. How an endpoint handles the loss of the frame depends on the type of the frame. Some frames are simply retransmitted, some have their contents moved to new frames, and others are never retransmitted.

When a packet is detected as lost, the sender re-sends any frames as necessary:

- All application data sent in STREAM frames MUST be retransmitted, unless the endpoint has sent a RST_STREAM for that stream. When an endpoint sends a RST_STREAM frame, data outstanding on that stream SHOULD NOT be retransmitted, since subsequent data on this stream is expected to not be delivered by the receiver.

- ACK and PADDING frames MUST NOT be retransmitted. ACK frames are cumulative, so new frames containing updated information will be sent as described in Section 8.2.

- All other frames MUST be retransmitted.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [QUIC-RECOVERY].

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been queued (but not necessarily delivered to the application). This also means that any stream state transitions triggered by STREAM or RST_STREAM frames have occurred. Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet.

To avoid creating an indefinite feedback loop, an endpoint MUST NOT generate an ACK frame in response to a packet containing only ACK or PADDING frames.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [QUIC-RECOVERY].

9.1. Special Considerations for PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 ([RFC1191] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

As a result, endpoints that implement PMTUD in IPv4 SHOULD take steps to mitigate this risk. For instance, an application could:

- Set the IPv4 Don't Fragment (DF) bit on a small proportion of packets, so that most invalid ICMP messages arrive when there are no DF packets outstanding, and can therefore be identified as spurious.
- Store additional information from the IP or UDP headers from DF packets (for example, the IP ID or UDP checksum) to further authenticate incoming Datagram Too Big messages.
- Any reduction in PMTU due to a report contained in an ICMP packet is provisional until QUIC's loss detection algorithm determines that the packet is actually lost.

10. Streams: QUIC's Data Structuring Abstraction

Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction modeled closely on HTTP/2 streams [RFC7540].

Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled.

Data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation.

The creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection.

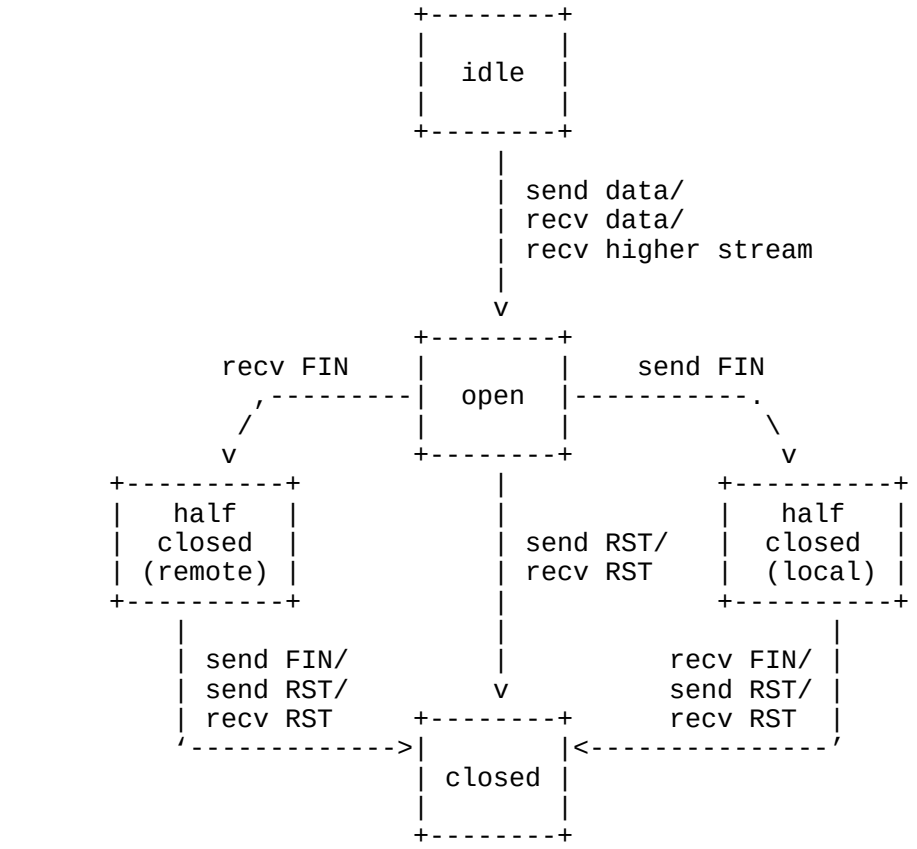
Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure.

An alternative view of QUIC streams is as an elastic "message" abstraction, similar to the way ephemeral streams are used in SST [SST], which may be a more appealing description for some applications.

10.1. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the possibility of out-of-order delivery due to the use of multiple

streams in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



send: endpoint sends this frame
recv: endpoint receives this frame

data: application data in a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame

Figure 11: Lifecycle of a stream

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. For the purpose of state transitions, the FIN flag is processed as a separate event to the frame that bears it; a STREAM frame with the FIN flag set can cause two state transitions. When the FIN flag is sent on an empty

STREAM frame, the offset in the STREAM frame MUST be one greater than the last data byte sent on this stream.

The recipient of a frame which changes stream state will have a delayed view of the state of a stream while the frame is in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing. Endpoints can use acknowledgments to understand the peer's subjective view of stream state at any given time.

Streams have the following states:

10.1.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM frame causes the stream to become "open". The stream identifier is selected as described in Section 10.2. The same STREAM frame can also cause a stream to immediately become "half-closed".

Receiving a STREAM frame on a peer-initiated stream (that is, a packet sent by a server on an even-numbered stream or a client packet on an odd-numbered stream) also causes all lower-numbered "idle" streams in the same direction to become "open". This could occur if a peer begins sending on streams in a different order to their creation, or it could happen if packets are lost or reordered in transit.

Receiving any frame other than STREAM or RST_STREAM on a stream in this state MUST be treated as a connection error (Section 12) of type YYYY.

10.1.2. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, a sending peer must observe the flow-control limit advertised by its receiving peer (Section 11).

From this state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an FIN flag causes the stream state to become "half-closed (local)". An endpoint receiving a FIN flag causes the stream state to become "half-closed (remote)" once

all preceding data has arrived. The receiving endpoint MUST NOT consider the stream state to have changed until all data has arrived.

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

10.1.3. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending STREAM frames; WINDOW_UPDATE and RST_STREAM MAY be sent in this state.

A stream transitions from this state to "closed" when a STREAM frame that contains a FIN flag is received and all prior data has arrived, or when either peer sends a RST_STREAM frame.

An endpoint that closes a stream MUST NOT send data beyond the final offset that it has chosen, see Section 10.1.5 for details.

An endpoint can receive any type of frame in this state. Providing flow-control credit using WINDOW_UPDATE frames is necessary to continue receiving flow-controlled frames. In this state, a receiver MAY ignore WINDOW_UPDATE frames for this stream, which might arrive for a short period after a frame bearing the FIN flag is sent.

10.1.4. half-closed (remote)

A stream that is "half-closed (remote)" is no longer being used by the peer to send any data. In this state, a sender is no longer obligated to maintain a receiver stream-level flow-control window.

A stream that is in the "half-closed (remote)" state will have a final offset for received data, see Section 10.1.5 for details.

A stream in this state can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level and connection-level flow-control limits (Section 11).

A stream can transition from this state to "closed" by sending a frame that contains a FIN flag or when either peer sends a RST_STREAM frame.

10.1.5. closed

The "closed" state is the terminal state.

An endpoint will learn the final offset of the data it receives on a stream when it enters the "half-closed (remote)" or "closed" state. The final offset is carried explicitly in the RST_STREAM frame; otherwise, the final offset is the offset of the end of the data carried in STREAM frame marked with a FIN flag.

An endpoint MUST NOT send data on a stream at or beyond the final offset.

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a QUIC_STREAM_DATA_AFTER_TERMINATION error (see Section 12). A receiver SHOULD treat receipt of data at or beyond the final offset as a QUIC_STREAM_DATA_AFTER_TERMINATION error. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

An endpoint that receives a RST_STREAM frame (and which has not sent a FIN or a RST_STREAM) MUST immediately respond with a RST_STREAM frame, and MUST NOT send any more data on the stream. This endpoint may continue receiving frames for the stream on which a RST_STREAM is received.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM frame might have already sent - or enqueued for sending - frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

STREAM frames received after sending RST_STREAM are counted toward the connection and stream flow-control windows. Even though these frames might be ignored, because they are sent before their sender receives the RST_STREAM, the sender will consider the frames to count against its flow-control windows.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section 12). Frames of unknown types are ignored.

(TODO: QUIC_STREAM_NO_ERROR is a special case. Write it up.)

10.2. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the StreamID. To avoid StreamID collision, clients MUST initiate streams usinge odd-numbered StreamIDs; streams initiated by the server MUST use even-numbered StreamIDs.

A StreamID of zero (0x0) is reserved and used for connection-level flow control frames (Section 11); the StreamID of zero cannot be used to establish a new stream.

StreamID 1 (0x1) is reserved for the cryptographic handshake. StreamID 1 MUST NOT be used for application data, and MUST be the first client-initiated stream.

A QUIC endpoint cannot reuse a StreamID on a given connection. Streams MUST be created in sequential order. Open streams can be used in any order. Streams that are used out of order result in lower-numbered streams in the same direction being counted as open.

All streams, including stream 1, count toward this limit. Thus, a concurrent stream limit of 0 will cause a connection to be unusable. Application protocols that use QUIC might require a certain minimum number of streams to function correctly. If a peer advertises an concurrent stream limit (concurrent_streams) that is too small for the selected application protocol to function, an endpoint MUST terminate the connection with an error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.3. Stream Concurrency

An endpoint limits the number of concurrently active incoming streams by setting the concurrent stream limit (see Section 7.3.1) in the transport parameters. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the concurrent stream limit.

A recently closed stream MUST also be considered to count toward this limit until packets containing all frames required to close the stream have been acknowledged. For a stream which closed cleanly, this means all STREAM frames have been acknowledged; for a stream

which closed abruptly, this means the RST_STREAM frame has been acknowledged.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.4. Sending and Receiving Data

Once a stream is created, endpoints may use the stream to send and receive data. Each endpoint may send a series of STREAM frames encapsulating data on a stream until the stream is terminated in that direction. Streams are an ordered byte-stream abstraction, and they have no other structure within them. STREAM frame boundaries are not expected to be preserved in retransmissions from the sender or during delivery to the application at the receiver.

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. The first byte of data that is sent on a stream has the stream offset 0. The largest offset delivered on a stream MUST be less than 2^64. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

The cryptographic handshake stream, Stream 1, MUST NOT be subject to congestion control or connection-level flow control, but MUST be subject to stream-level flow control. An endpoint MUST NOT send data on any other stream without consulting the congestion controller and the flow controller.

Flow control is described in detail in Section 11, and congestion control is described in the companion document [QUIC-RECOVERY].

10.5. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [RFC7540], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging priotization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to

define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. When deciding which streams to dedicate resources to, QUIC SHOULD use the information provided by the application. Failure to account for priority of streams can result in suboptimal performance.

Stream priority is most relevant when deciding which stream data will be transmitted. Often, there will be limits on what can be transmitted as a result of connection flow control or the current congestion controller state.

Giving preference to the transmission of its own management frames ensures that the protocol functions efficiently. That is, prioritizing frames other than STREAM frames ensures that loss recovery, congestion control, and flow control operate effectively.

Stream 1 MUST be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

STREAM frames that are determined to be lost SHOULD be retransmitted before sending new data, unless application priorities indicate otherwise. Retransmitting lost STREAM frames can fill in gaps, which allows the peer to consume already received data and free up flow control window.

11. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. This section describes QUIC's flow-control mechanisms.

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [RFC7540]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i) Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow

control, which prevents a single stream from consuming the entire receive buffer for a connection.

A receiver sends WINDOW_UPDATE frames to the sender to advertise additional credit by sending the absolute byte offset in the stream or in the connection which it is willing to receive.

The initial flow control credit is 65536 bytes for both the stream and connection flow controllers.

A receiver MAY advertise a larger offset at any point in the connection by sending a WINDOW_UPDATE frame. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset via a WINDOW_UPDATE frame, it MUST NOT subsequently advertise a smaller offset. A sender may receive WINDOW_UPDATE frames out of order; a sender MUST therefore ignore any WINDOW_UPDATE that does not move the window forward.

A receiver MUST close the connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error (Section 12) if the peer violates the advertised stream or connection flow control windows.

A sender MUST send BLOCKED frames to indicate it has data to write but is blocked by lack of connection or stream flow control credit. BLOCKED frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

A receiver advertises credit for a stream by sending a WINDOW_UPDATE frame with the StreamID set appropriately. A receiver may use the current offset of data consumed to determine the flow control offset to be advertised. A receiver MAY send copies of a WINDOW_UPDATE frame in multiple packets in order to make sure that the sender receives it before running out of flow control credit, even if one of the packets is lost.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames on all streams contributing to connection flow control. A receiver advertises credit for a connection by sending a WINDOW_UPDATE frame with the StreamID set to zero (0x00). A receiver maintains a cumulative sum of bytes received on all streams contributing to connection-level flow control, to check for flow control violations. A receiver may maintain a cumulative sum of bytes consumed on all contributing streams to determine the connection-level flow control offset to be advertised.

11.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives. Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a WINDOW_UPDATE which will never come.

11.1.1. Mid-stream RST_STREAM

On receipt of a RST_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. This could result in the endpoints getting out of sync, since the RST_STREAM frame may have arrived out of order and there may be further bytes in flight. The data sender would have counted the data against its connection level flow control budget, but a receiver that has not received these bytes would not know to include them as well. The receiver must learn the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

11.1.2. Response to a RST_STREAM

Since streams are bidirectional, a sender of a RST_STREAM needs to know how many bytes the peer has sent on the stream. If an endpoint receives a RST_STREAM frame and has sent neither a FIN nor a RST_STREAM, it MUST send a RST_STREAM in response, bearing the offset of the last byte sent on this stream as the final offset.

11.1.3. Offset Increment

This document leaves when and how many bytes to advertise in a WINDOW_UPDATE to the implementation, but offers a few considerations. WINDOW_UPDATE frames constitute overhead, and therefore, sending a WINDOW_UPDATE with small offset increments is undesirable. At the same time, sending WINDOW_UPDATES with large offset increments requires the sender to commit to that amount of buffer.

Implementations must find the correct tradeoff between these sides to determine how large an offset increment to send in a WINDOW_UPDATE.

A receiver MAY use an autotuning mechanism to tune the size of the offset increment to advertise based on a roundtrip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

11.1.4. BLOCKED frames

If a sender does not receive a WINDOW_UPDATE frame when it has run out of flow control credit, the sender will be blocked and MUST send a BLOCKED frame. A BLOCKED frame is expected to be useful for debugging at the receiver. A receiver SHOULD NOT wait for a BLOCKED frame before sending a WINDOW_UPDATE, since doing so will cause at least one roundtrip of quiescence. For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibiity of loss, a receiver should send a WINDOW_UPDATE frame at least two roundtrips before it expects the sender to get blocked.

12. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 12.1), or a single stream (see Section 12.2).

The most appropriate error code (Section 12.3) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

Public Reset is not suitable for any error that can be signaled with a CONNECTION_CLOSE or RST_STREAM frame. Public Reset MUST NOT be sent by an endpoint that has the state necessary to send a frame on the connection.

12.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE frame (Section 8.8). An endpoint MAY close the connection in this manner, even if the error only affects a single stream.

A CONNECTION_CLOSE frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing a CONNECTION_CLOSE frame if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing CONNECTION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to send a Public Reset packet.

12.2. Stream Errors

If the error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can sent a RST_STREAM frame (Section 8.5) with an appropriate error code to terminate just the affected stream.

Stream 1 is critical to the functioning of the entire connection. If stream 1 is closed with either a RST_STREAM or STREAM frame bearing the FIN flag, an endpoint MUST generate a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

Some application protocols make other streams critical to that protocol. An application protocol does not need to inform the transport that a stream is critical; it can instead generate appropriate errors in response to being notified that the critical stream is closed.

An endpoint MAY send a RST_STREAM frame in the same packet as a CONNECTION_CLOSE frame.

12.3. Error Codes

Error codes are 32 bits long, with the first two bits indicating the source of the error code:

0x00000000-0x3FFFFFFF: Application-specific error codes. Defined by each application-layer protocol.

0x40000000-0x7FFFFFFF: Reserved for host-local error codes. These codes MUST NOT be sent to a peer, but MAY be used in API return codes and logs.

0x80000000-0xBFFFFFFF: QUIC transport error codes, including packet protection errors. Applicable to all uses of QUIC.

0xC0000000-0xFFFFFFFF: Cryptographic error codes. Defined by the cryptographic handshake protocol in use.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE or RST_STREAM frame. Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

QUIC_INTERNAL_ERROR (0x80000001): Connection has reached an invalid state.

QUIC_STREAM_DATA_AFTER_TERMINATION (0x80000002): There were data frames after the a fin or reset.

QUIC_INVALID_PACKET_HEADER (0x80000003): Control frame is malformed.

QUIC_INVALID_FRAME_DATA (0x80000004): Frame data is malformed.

QUIC_MULTIPLE_TERMINATION_OFFSETS (0x80000005): Multiple final offset values were received on the same stream

QUIC_STREAM_CANCELLED (0x80000006): The stream was cancelled

QUIC_CLOSED_CRITICAL_STREAM (0x80000007): A stream that is critical to the protocol was closed.

QUIC_MISSING_PAYLOAD (0x80000030): The packet contained no payload.

QUIC_INVALID_STREAM_DATA (0x8000002E): STREAM frame data is malformed.

QUIC_UNENCRYPTED_STREAM_DATA (0x8000003D): Received STREAM frame data is not encrypted.

QUIC_MAYBE_CORRUPTED_MEMORY (0x80000059): Received a frame which is likely the result of memory corruption.

QUIC_INVALID_RST_STREAM_DATA (0x80000006): RST_STREAM frame data is malformed.

QUIC_INVALID_CONNECTION_CLOSE_DATA (0x80000007): CONNECTION_CLOSE frame data is malformed.

QUIC_INVALID_GOAWAY_DATA (0x80000008): GOAWAY frame data is malformed.

QUIC_INVALID_WINDOW_UPDATE_DATA (0x80000039): WINDOW_UPDATE frame data is malformed.

QUIC_INVALID_BLOCKED_DATA (0x8000003A): BLOCKED frame data is malformed.

QUIC_INVALID_PATH_CLOSE_DATA (0x8000004E): PATH_CLOSE frame data is malformed.

QUIC_INVALID_ACK_DATA (0x80000009): ACK frame data is malformed.

QUIC_INVALID_VERSION_NEGOTIATION_PACKET (0x8000000A): Version negotiation packet is malformed.

QUIC_INVALID_PUBLIC_RST_PACKET (0x8000000b): Public RST packet is malformed.

QUIC_DECRYPTION_FAILURE (0x8000000c): There was an error decrypting.

QUIC_ENCRYPTION_FAILURE (0x8000000d): There was an error encrypting.

QUIC_PACKET_TOO_LARGE (0x8000000e): The packet exceeded kMaxPacketSize.

QUIC_PEER_GOING_AWAY (0x80000010): The peer is going away. May be a client or server.

QUIC_INVALID_STREAM_ID (0x80000011): A stream ID was invalid.

QUIC_INVALID_PRIORITY (0x80000031): A priority was invalid.

QUIC_TOO_MANY_OPEN_STREAMS (0x80000012): Too many streams already open.

QUIC_TOO_MANY_AVAILABLE_STREAMS (0x8000004c): The peer created too many available streams.

QUIC_PUBLIC_RESET (0x80000013): Received public reset for this connection.

QUIC_INVALID_VERSION (0x80000014): Invalid protocol version.

QUIC_INVALID_HEADER_ID (0x80000016): The Header ID for a stream was too far from the previous.

QUIC_INVALID_NEGOTIATED_VALUE (0x80000017): Negotiable parameter received during handshake had invalid value.

QUIC_DECOMPRESSION_FAILURE (0x80000018): There was an error decompressing data.

QUIC_NETWORK_IDLE_TIMEOUT (0x80000019): The connection timed out due to no network activity.

QUIC_HANDSHAKE_TIMEOUT (0x80000043): The connection timed out waiting for the handshake to complete.

QUIC_ERROR_MIGRATING_ADDRESS (0x8000001a): There was an error encountered migrating addresses.

QUIC_ERROR_MIGRATING_PORT (0x80000056): There was an error encountered migrating port only.

QUIC_EMPTY_STREAM_FRAME_NO_FIN (0x80000032): We received a STREAM_FRAME with no data and no fin flag set.

QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA (0x8000003b): The peer received too much data, violating flow control.

QUIC_FLOW_CONTROL_SENT_TOO_MUCH_DATA (0x8000003f): The peer sent too much data, violating flow control.

QUIC_FLOW_CONTROL_INVALID_WINDOW (0x80000040): The peer received an invalid flow control window.

QUIC_CONNECTION_IP_POOLED (0x8000003e): The connection has been IP pooled into an existing connection.

QUIC_TOO_MANY_OUTSTANDING_SENT_PACKETS (0x80000044): The connection has too many outstanding sent packets.

QUIC_TOO_MANY_OUTSTANDING_RECEIVED_PACKETS (0x80000045): The connection has too many outstanding received packets.

QUIC_CONNECTION_CANCELLED (0x80000046): The QUIC connection has been cancelled.

QUIC_BAD_PACKET_LOSS_RATE (0x80000047): Disabled QUIC because of high packet loss rate.

QUIC_PUBLIC_RESETS_POST_HANDSHAKE (0x80000049): Disabled QUIC because of too many PUBLIC_RESETs post handshake.

QUIC_TIMEOUTS_WITH_OPEN_STREAMS (0x8000004a): Disabled QUIC because of too many timeouts with streams open.

QUIC_TOO_MANY_RTOS (0x80000055): QUIC timed out after too many RTOs.

QUIC_ENCRYPTION_LEVEL_INCORRECT (0x8000002c): A packet was received with the wrong encryption level (i.e. it should have been encrypted but was not.)

QUIC_VERSION_NEGOTIATION_MISMATCH (0x80000037): This connection involved a version negotiation which appears to have been tampered with.

QUIC_IP_ADDRESS_CHANGED (0x80000050): IP address changed causing connection close.

QUIC_ADDRESS_VALIDATION_FAILURE (0x80000051): Client address validation failed.

QUIC_TOO_MANY_FRAME_GAPS (0x8000005d): Stream frames arrived too discontiguously so that stream sequencer buffer maintains too many gaps.

QUIC_TOO_MANY_SESSIONS_ON_SERVER (0x80000060): Connection closed because server hit max number of sessions allowed.

13. Security and Privacy Considerations

13.1. Spoofed ACK Attack

An attacker receives an STK from the server and then releases the IP address on which it received the STK. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker then spoofs ACK frames to the server which cause the server to potentially drown the victim in data.

There are two possible mitigations to this attack. The simplest one is that a server can unilaterally create a gap in packet-number space. In the non-attack scenario, the client will send an ACK frame with the larger value for largest acknowledged. In the attack scenario, the attacker could acknowledge a packet in the gap. If the server sees an acknowledgment for a packet that was never sent, the connection can be aborted.

The second mitigation is that the server can require that acknowledgments for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is encrypted with a forward-secure key,

then any acknowledgments that are received for them MUST also be forward-secure encrypted. Since the attacker will not have the forward secure key, the attacker will not be able to generate forward-secure encrypted packets with ACK frames.

14. IANA Considerations

14.1. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC Protocol" heading.

The "QUIC Transport Parameters" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC5226]. Values with the first byte 0xff are reserved for Private Use [RFC5226].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Parameter Name: A short mnemonic for the parameter.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. The expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are shown in Table 4.

Value	Parameter Name	Specification
0x0000	stream_fc_offset	Section 7.3.1
0x0001	connection_fc_offset	Section 7.3.1
0x0002	concurrent_streams	Section 7.3.1
0x0003	idle_timeout	Section 7.3.1
0x0004	truncate_connection_id	Section 7.3.1

Table 4: Initial QUIC Transport Parameters Entries

15. References

15.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC".

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <http://www.rfc-editor.org/info/rfc1191>.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <http://www.rfc-editor.org/info/rfc1981>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <http://www.rfc-editor.org/info/rfc4821>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

15.2. Informative References

[EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <https://goo.gl/dMVtFi>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <http://www.rfc-editor.org/info/rfc2360>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[SST] Ford, B., "Structured Streams: A New Transport Abstraction", DOI 10.1145/1282427.1282421, ACM SIGCOMM Computer Communication Review Volume 37 Issue 4, October 2007.

15.3. URIs

[1] https://github.com/quicwg/base-drafts/wiki/QUIC-Versions

Appendix A. Contributors

The original authors of this specification were Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk.

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN]. In alphabetical order, the contributors to the pre-IETF QUIC project at Google are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

Appendix B. Acknowledgments

Special thanks are due to the following for helping shape pre-IETF QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund.

This document has benefited immensely from various private discussions and public ones on the quic@ietf.org and proto-quic@chromium.org mailing lists. Our thanks to all.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-transport-01:

- Defined short and long packet headers (#40, #148, #361)
- Defined a versioning scheme and stable fields (#51, #361)
- Define reserved version values for "greasing" negotiation (#112, #278)
- The initial packet number is randomized (#35, #283)
- Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)

- Defined client address validation (#52, #118, #120, #275)
- Define transport parameters as a TLS extension (#122)
- SCUP and COPT parameters are no longer valid (#116, #117)
- Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- The server chooses connection IDs in its final flight (#119, #349, #361)
- The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- Definied a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- Path MTU Discovery (#64, #106)
- The initial handshake packet from the client needs to fit in a single packet (#338)
- Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- Require that frames are processed when packets are acknowledged (#381, #341)
- Removed the STOP_WAITING frame (#66)
- Don’t require retransmission of old timestamps for lost ACK frames (#308)
- Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- Error handling definitions (#335)
- Split error codes into four sections (#74)
- Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- Define packet protection rules (#336)

- Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RST_STREAM, before it closes (#381)
- Remove stream reservation from state machine (#174, #280)
- Only stream 0 does not contributing to connection-level flow control (#204)
- Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- Remove connection-level flow control exclusion for some streams (except 1) (#246)
- RST_STREAM affects connection-level flow control (#162, #163)
- Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- Added the ability to pad between frames (#158, #276)
- Remove error code and reason phrase from GOAWAY (#352, #355)
- GOAWAY includes a final stream number for both directions (#347)
- Error codes for RST_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- Defined priority as the responsibility of the application protocol (#104, #303)

C.2. Since draft-ietf-quic-transport-00:

- Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- Defined versioning
- Reworked description of packet and frame layout
- Error code space is divided into regions for each component
- Use big endian for all numeric values

C.3. Since draft-hamilton-quic-transport-protocol-01:

- Adopted as base for draft-ietf-quic-tls.
- Updated authors/editors list.
- Added IANA Considerations section.
- Moved Contributors and Acknowledgments to appendices.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 25, 2017

I. Johansson
Ericsson AB

February 21, 2017

ECN support in QUIC
draft-johansson-quic-ecn-01

Abstract

This memo outlines the ECN support in QUIC. The intention is that most of the material ends up updating other new or existing QUIC protocol specifications, thus it may be possible that this draft does not warrant a working group status.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 25, 2017.

Copyright Notice

Table of Contents

1. Introduction

ECN support in transport protocols is a fundamental feature that should be included in the QUIC specification as a mandatory element. The benefits of ECN is described in [I-D.ietf-aqm-ecn-benefits]. The ECN support should be implemented to support both present and future ECN, the latter is outlined in [I-D.ietf-tsvwg-ecn-experimentation], of particular interest is the ability to discriminate between classic ECN and L4S ECN by means of differentiation between the use of the ECT(0) and ECT(1) code points. This draft does however not delve into the details of the congestion control implementation.

2. Elements of ECN support

This draft covers the following aspects of ECN support:

- ECN negotiation
- ECN echo
- ECN bits in the IP header, semantics
- Fallback in case of ECN fault

- OS socket specifics, access to the ECN bits
- Monitoring

2.1. ECN negotialtion

ECN support in QUIC needs to be negotiated. The reasons is that network elements may not support ECN and may either clear the ECN bits or simply discard packets that have the ECN bits set. In addition, a QUIC implementation may not have access to the ECN bits in the IP header due to OS dependent restrictions, investigations (Piers O'Hanlon) have indicated that this is in certain cases an asymmetric property, for instance while it is possible to set the ECN bits it is not possible to read them.

It is also required that the ECN negotiation does not interfere with the connection setup, in other words a failed ECN negotation should not cause an extra roundtrip for the connection setup.

The suggested method in this draft is to add an ECN negotiation frame that is transmitted when connection setup is completed. Both peers MUST transmit the ECN negotation frame. The ECN negotiation frame is shown below.

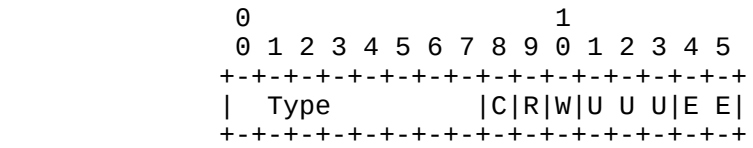


Figure 1: ECN negotation frame

The 2nd byte contains the flags:

- C: Challenge bit, indicates that the transmitted ECN negotiation frame is a challenge, if bit is not set then it is a response.
- R: Possible to read ECN bits in IP header
- W: Possible to write ECN bits in IP header
- EE : Echo of ECN bits
- U: Unused

A peer transmits the ECN negotiation frame with the R,W and EE bits in the 2nd byte set to '0' and the C bit set to '1'. This frame is echoed back with the flags set occording to the degree of ECN support

and with the ECN bits in the IP header of the received ECN negotiation frame copied to the EE field, the C bit is '0'. As both peers MUST transmit an ECN negotation frame there will be a total of 4 ECN negotiation frames transmitted, two challenges and two responses.

The IP header for the ECN negotiation frame should set the ECN bits to CE '11'. When the corresponding response is received then an EE pattern of '11' indicates that ECN is likely supported in the network. This does not give a full quarantee that ECN is supported in the network. Monitoring of the ECN field in the ACK-frame serves to give further indication of ECN support once ECN is turned on.

A peer is not allowed to set ECT on outgoing data packets until a ECN negotiation response that inticates that ECN is supported is received. In other words it is only the ECN negotiation frame that is allowed to set the ECN bits in the IP header.

A lack of an ECN negotiation response may indicate that the ECN challenge frame or the ECN response frame was lost or that a node in the network deliberately discards ECN-CE marked packets. The peer can transmit additonal ECN challeges with given time intervals to rule out accidentail packet loss. The detailed timing for this is T.B.D.

The mode mechanism in [RFC6679] can serve as in input to a solution for the support of ECN in the case that OS ECN support is asymmetric. It is however unclear how a QUIC implementation can determine asymmetric ECN support in the underlying OS. For instance the method to send ECN marked packets to the local host to determine OS support does not reveal if the OS ECN support is asymmetric.

2.2. ECN bits in the IP header, semantics

The ECN bits in the IP header should be set according to the recommendations in [I-D.ietf-tsvwg-ecn-experimentation]. This means that the meaning of ECT(0) and ECT(1) differ.

2.3. ECN echo

The ECN echo should prefferably go into the ACK frame [I-D.ietf-quic-transport], this is beneficial as the ECN information can then use some of the already existing data in the ACK frame for improved efficiency, this applies especially to alternatives 1 and 2 below. It is suggested that the 'U' bit in the ACK frame type is renamed 'E' to indicate the presence of an ECN field in the ACK frame, this makes it possible to omit the ECN information for the cases where ECN is not supported for the connection.

Currently there are three alternatives how to add ECN support to the ACK frames .

The first alternative inserts a one octet field that contains a 2 bit ECN echo, followed by the ACK block length. The ACK block length then dictates the number of received contiguous frames with the indicated ECN echo.

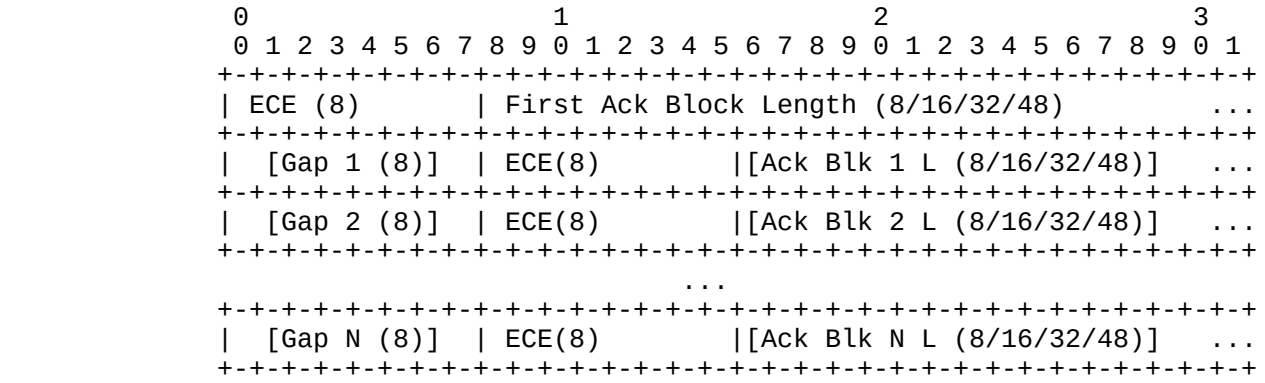


Figure 2: ECN field in ACK frame ACK block, alt 1

The second alternative encodes a variable length field that contains the ECN echoes for the frames listed in the ACK blocks. The length of the field is inferred from the ACK block lengths. No ECN echoes are indicated for the gaps (it is, after all, impossible to indicate status of the ECN bits for lost packets). For instance if the ACK blocks list 10 frames, then the length of the ECN echo field becomes 2*10=20bits, with additional 4 bits of padding the ECN echo field will then become 3 octets long.

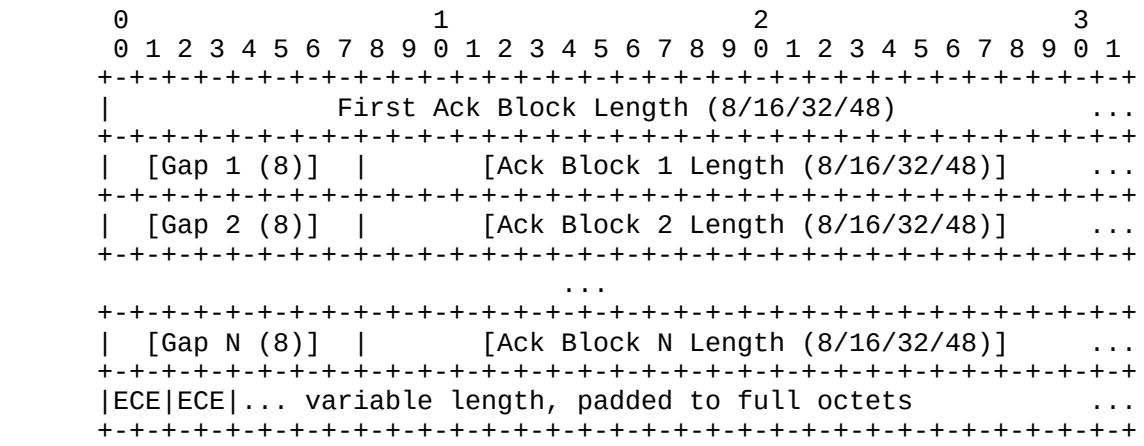


Figure 3: ECN field in ACK frame ACK block, alt 2

The third alternative encodes the number of bytes that are marked ECT(0), ECT(1) and CE with 32 bits each, the total extra overhead is thus 12 octets.

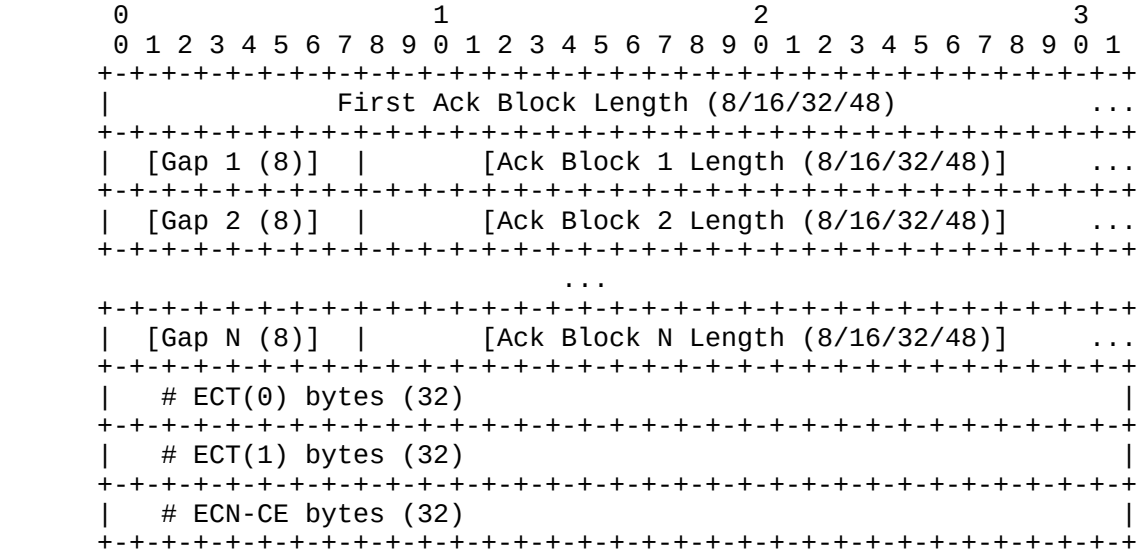


Figure 4: ECN field in ACK frame ACK block, alt 3

The fourth alternative use an extra byte to encode how many bits that encode each of the ECT/CE fields.

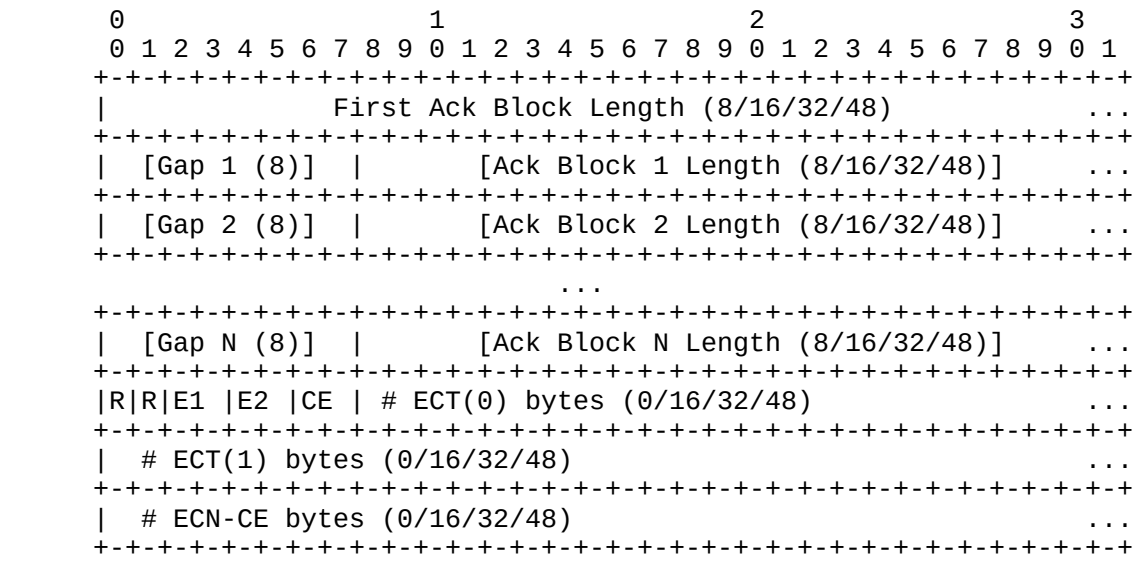


Figure 5: ECN field in ACK frame ACK block, alt 4

The E1,E2 and CE fields indicate the length of each encoding for the number of ECT(0), ECT(1) and ECN-CE marked bytes. This is encoded as:

- 00: 0 bits
- 01: 16 bits
- 10: 32 bits
- 11: 48bits

R indicates reserved bits.

There are pros an cons with the four alternatives:

- Alt 1: Is very compact in the case where the ECN bits are largely unchanged. However in the worst case where received frames flip forth and back between ECT and CE then each frame will require at least 3 octets overhead (ECE, ACK block length, Gap).
- Alt 2: Is quite compact as it only requires two bits encoding per frame. The additional overhead amounts to ceil(N*2/8) octets where the N is the sum of the ACK block lengths. On the downside is that it is a less efficient format for the case that the ECN

bits are unchanged. One uncertainty is if STOP_WAITING frames could make this encoding bulky.

- Alt 3: Has a fixed 12 octet overhead which may be beneficial as it gives a deterministic overhead. The possible drawback is that it is not possible to know exactly which frames have been remarked, something that can limit the ability to detect network ECN faults based on the method to transmit a pattern on ECT and CE marked packets.

- Alt 4: Is a variation to Alt 3 but has a variable length encoding that should consume less space, especially in the cases that one of the ECT code points is not used and for the case that packets are only sporadically ECN-CE marked. This alternative also makes it unnecessary to use a bit in the ACK frame type to indicate the precense of an ECN field as this can be indicated in a efficient way with the one byte header in this format. E0=E1=CE = 00 indicates that the following ECT and CE fields are encoded with zero bits.

Which of the three formats above (or something else) that is the best alternative is subject to discussion.

2.4. Fallback in case of ECN fault

ECN can be subject to issues in network equipment, such as remarking to Not-ECN, remarking from ECT(0) to ECT(1) and vice versa or constant remarking to ECN-CE. Furthermore ECT marked packets may be discarded in the network. While these problems seem to be rare, see for instance [McQuistin-Perkins], it is still necessary to safeguard against such problems.

A peer should disable ECN for its outgoing packets if ECN fault is detected, it is however still possible for the other peer to use ECN.

TODO add more information as regards to how to detect network ECN faults. [ECN-fallback](expired) gives a few examples for fault detection. Examples on how to detect ECN faults include for instance the method to set ECT and CE for outgoing packets according to a given pattern.

Fallback in case of ECN faults is not an issue only for QUIC, it is here suggested that mechanisms for this is described in a non QUIC related draft, for instance in TSVWG.

2.5. OS socket specifics, access to the ECN bits

ECN support in QUIC comes with the additional challenge that it is necessary to somehow access the ECN bits in the IP headers. In TCP this is provided without major concerns as TCP is generally implemented in OS kernel space. QUIC can however be implemented both in user space or kernel space and is layered on top of UDP, which means that access to the ECN bits is not a given, instead various tricks are needed.

The text below is copy-pasted from [OHanlon].

"To set ECN on Linux, BSD and OSX one can use IP_TOS socket option, with the setsockopt() call, to set the relevant ECN bits of the TOS byte. On Windows one can use a similar technique though firstly one has to enable TOS byte setting by enabling a particular Registry key (DisableUserTOSSetting=0 (see https://msdn.microsoft.com/en-us/library/windows/desktop/dd874008%28v=vs.85%29.aspx One could also probably use the libpcap write functionality."

"To obtain the ECN bits from a packet one needs a mechanism to retrieve the ECN bits from each packet. On Linux, one needs to firstly set the IP_RECVTOS socket option on the receiving socket, and use the recvmsg() call to receive a packet, and then retrieve the TOS byte from the associated csmg structure returned by the recvmsg() call. This still works with linux-4.2.3. On OSX/BSD there are no suitable socket options to retrieve the ECN/TOS bits and one cannot use raw sockets as they do not function for UDP/TCP sockets (they do work with ICMP), so one has to use alternatives such the bpf interface, or a REDIRECT socket. Whilst on Windows it seems that the only way to retrieve the ECN bits is via a raw socket, or custom NDIS driver, though it’s possible there’s an API I’m missing."

TODO: Write a more detailed description on how to implement ECN support in QUIC for different OS stacks.

2.6. Monitoring

A QUIC implementation should monitor the ECN functionality in order to provide input to e.g. service providers to improve ECN support in the networks. Items of interest are:

- Black holes, ECT or CE marked packets are discarded.
- Faulty remarking, e.g. ECT(0) is remarked to ECT(1) or Not-ECT.
- Continuous CE marking, possible indication of faulty on/off ECN marking, but can also be an effect of severe congestion.

- Degree of L4S support. L4S should generally give low queue latency. Estimation of one way queue delay for L4S enabled QUIC connections can be used to determine if there are congested nodes along the path that are not L4S capable.

3. IANA Considerations

T.B.D.

4. Open questions

A list of open questions:

- Is it sufficient that one peer sends an ECN negotiation challenge frame?.
- Should the ECN field in the ACK frame be mandatory ? (in which case it is not necessary to indicate its presence)
- Should all packets be ECT or should there be special patterns to improve fault detection.
- Write up a more detailed description on how to implement ECN support in QUIC for different OS stacks.
- Determine which ECN echo encoding in the ACK frame is the best alternative.
- Is a completely new ACK frame an alternative ?
- How do STOP_WAITING frames affect the ECN echo overhead.
- Outline possible connection migration actions
- Are there any security implications with the smalle ECN negotiation frame ?

5. Security Considerations

T.B.D

6. Acknowledgements

The following persons have contributed with comments and suggestions for improvements: Mirja Kuehlewind, Koen De Schepper, Piers O'Hanlon, Michael Welzl, Marcelo Bagnulo Braun, Martin Duke

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[Bagnulo] "Adding Explicit Congestion Notification (ECN) to TCP control packets and TCP retransmissions", <https://tools.ietf.org/id/draft-bagnulo-tcpm-generalized-ecn-00.txt>.

[ECN-fallback] "A Mechanism for ECN Path Probing and Fallback", <https://www.ietf.org/archive/id/draft-kuehlewind-tcpm-ecn-fallback-01.txt>.

[I-D.ietf-aqm-ecn-benefits] Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", draft-ietf-aqm-ecn-benefits-08 (work in progress), November 2015.

[I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[I-D.ietf-tsvwg-ecn-experimentation] Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-00 (work in progress), December 2016.

[McQuistin-Perkins] ""Is Explicit Congestion Notification usable with UDP?", Proceedings of the ACM Internet Measurement Conference, Tokyo, Japan, October 2015. DOI:10.1145/2815675.2815716", <https://csperkins.org/publications/2015/10/mcquistin2015ecn-udp.pdf>.

[OHanlon] "ECN support in different OS stacks", <https://mailarchive.ietf.org/arch/msg/rmcat/rRKF3PVmFL2zHCp1bOPKimqSsbM>.

[RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <http://www.rfc-editor.org/info/rfc6679>.

[RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <http://www.rfc-editor.org/info/rfc6789>.

[RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <http://www.rfc-editor.org/info/rfc7560>.

Author's Address

Ingemar Johansson
Ericsson AB
Laboratoriegraend 11
Luleaa 977 53
Sweden

Phone: +46 730783289
Email: ingemar.s.johansson@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
March 08, 2017

Applicability of the QUIC Transport Protocol
draft-kuehlewind-quic-applicability-00

Abstract

This document discusses the applicability of the QUIC transport protocol, focusing on caveats impacting application protocol development and deployment over QUIC. Its intended audience is designers of application protocol mappings to QUIC, and implementors of these application protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 9, 2017.

Table of Contents

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http] such as stream-multiplexing to avoid head-of-line blocking. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. This means the version of QUIC that is currently under development will integrate TLS 1.3 [I-D.ietf-quic-tls] to encrypt all payload data and most header information.

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This includes general guidance for application use of HTTP/2 over QUIC as well as the use of other application layer protocols over QUIC. For specific guidance on how to integrate HTTP/2 with QUIC, see [I-D.ietf-quic-http].

In the following sections we discuss specific caveats to QUIC's applicability, and issues that application developers must consider when using QUIC as a transport for their application.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. The Necessity of Fallback

QUIC uses UDP as a substrate for userspace implementation and port numbers for NAT and middlebox traversal. While there is no evidence of widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [Edeline16], somewhere between three [Trammell16] and five [Swett16] percent of networks simply block UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol. This fallback SHOULD provide TLS 1.3 or equivalent cryptographic protection, if available, in order to keep fallback from being exploited as a downgrade attack. In the case of HTTP, this fallback is TLS 1.3 over TCP.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed. Further, TCP by default does not support 0-RTT session resumption. TCP Fast Open could be used, but might no be supported by the far end or could be blocked on the network path. Note that there is some evidence of middleboxes blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]). Moreover, while encryption (in this case TLS) is inseparable integrated with QUIC, TLS negotiation over TCP can be blocked. In case it is RECOMMENDED to abort the connection, allowing the application to present a suitable prompt to the user that secure communication is unavailable.

We hope that the deployment of a proposed standard version of the QUIC protocol will provide an incentive for these networks to permit QUIC traffic. Indeed, the ability to treat QUIC traffic statefully as discussed in section 3.1 of [draft-kuehlewind-quic-manageability] would remove one network management incentive to block this traffic.

3. Zero RTT: Here There Be Dragons

QUIC provides for 0-RTT connection establishment (see section 3.2 of [I-D.ietf-quic-transport]). However, data in the frames contained in the first packet of a such a connection must be treated specially by the application layer. Since a retransmission of these frames resulting from a lost acknowledgment may cause the data to appear twice, either the application-layer protocol has to be designed such that all such data is treated as idempotent, or there must be some application-layer mechanism for recognizing spuriously retransmitted frames and dropping them.

Applications that cannot treat data that may appear in a 0-RTT connection establishment as idempotent MUST NOT use 0-RTT establishment. For this reason the QUIC transport SHOULD provide an interface for the application to indicate if 0-RTT support is in general desired or a way to indicate if data is idempotent.

4. Stream versus Flow Multiplexing

QUIC's stream multiplexing feature allows applications to run multiple streams over a single connection, without head-of-line blocking between streams, associated at a point in time with a single five-tuple. Streams are meaningful only to the application; since stream information is carried inside QUIC's encryption boundary, no information about the stream(s) whose frames are carried by a given packet is visible to the network.

Stream multiplexing is not intended to be used for differentiating streams in terms of network treatment. Application traffic requiring different network treatment SHOULD therefore be carried over different five-tuples (i.e. multiple QUIC connections). Given QUIC's ability to send application data on the first packet of a connection (if a previous connection to the same host has been successfully established to provide the respective credentials), the cost for establishing another connection are extremely low.

[EDITOR'S NOTE: For discussion: If establishing a new connection does not seem to be sufficient, the protocol's rebinding functionality (see section 3.7 of [I-D.ietf-quic-transport]) could be extended to allow multiple five-tuples to share a connection ID simultaneously, instead of sequentially.]

5. Prioritization

Stream prioritization is not exposed to the network, nor to the receiver. Prioritization can be realized by the sender and the QUIC transport should provide and interface for applications to prioritize streams [I-D.ietf-quic-transport].

Priority handling of retransmissions may be implemented in the transport layer and [I-D.ietf-quic-transport] does not specify a specific way how this must be handled. Currently QUIC only provides fully reliable stream transmission, and as such prioritization of retransmission is likely beneficial. For not fully reliable streams priority scheduling of retransmissions over data of higher-priority streams might not be desired. In this case QUIC could also provide an interface or derive the prioritization decision from the reliability level of the stream.

6. Graceful connection closure

[EDITOR'S NOTE: give some guidance here about the steps an application should take; however this is still work in progress]

7. Information exposure and the Connection ID

QUIC exposed some information to the network in the unencrypted part of the header. This is either because there is no encryption context established yet or because this information is intended to be consumed by the network. Some of these information can be optionally exposed (still under discussion). Given that exposing these information can have privacy implications, an application may indicate to not support exposure of certain information.

In case of the connection ID this can be the case if the application has additional information that the client is not behind a NAT and the server is not behind a load balancer, and therefore it is unlikely that the addresses will be re-binded.

8. Use of Versions and Cryptographic Handshake

Versioning in QUIC may change the whole protocol behavior, beside some header fields that have been declared to be fixed. As such a new or higher version of QUIC does not necessarily provide a better service but just a very different service, an application needs to be able to select which versions of QUIC it wants to use.

The use of a different encryption scheme than TLS1.3 or higher needs a new version of QUIC. [I-D.ietf-quic-transport] specifies requirements for the cryptographic handshake as currently realized by TLS1.3 and described in a separate specification [I-D.ietf-quic-tls]. This split is performed to enable light-weight versioning with different cryptographic handshakes.

9. IANA Considerations

This document has no actions for IANA.

10. Security Considerations

See the security considerations in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well.

Application developers should note that any fallback they use when QUIC cannot be used due to network blocking of UDP SHOULD guarantee the same security properties as QUIC; if this is not possible, the

connection SHOULD fail to allow the application to explicitly handle fallback to a less-secure alternative. See Section 2.

11. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

12. References

12.1. Normative References

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

12.2. Informative References

[draft-kuehlewind-quic-manageability]
Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", March 2017.

[Edeline16]
Edeline, K., Kuehlewind, M., Trammell, B., Aben, E., and B. Donnet, "Using UDP for Internet Transport Evolution (arXiv preprint 1612.07816)", December 2016.

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[PaaschNanog]
Paasch, C., "Network Ssupport for TCP Fast Open (NANOG 67 presentation)", June 2016.

[Swett16] Swett, I., "QUIC Deployment Experience at Google (IETF96 QUIC BoF presentation)", July 2016.

[Trammell16]
Trammell, B. and M. Kuehlewind, "Internet Path Transparency Measurements using RIPE Atlas (RIPE72 MAT presentation)", May 2016.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
D. Druta
AT&T

March 09, 2017

Manageability of the QUIC Transport Protocol
draft-kuehlewind-quic-manageability-00

Abstract

This document discusses manageability of the QUIC transport protocol, focusing on caveats impacting network operations involving QUIC traffic. Its intended audience is network operators, as well as content providers that rely on the use of QUIC-aware middleboxes, e.g. for load balancing.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 10, 2017.

Table of Contents

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http]. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. The current version of QUIC integrates TLS [I-D.ietf-quic-tls] to encrypt all payload data and most header information. Given QUIC is an end-to-end transport protocol, all information in the protocol header, even that which can be inspected, is is not meant to be mutable by the network, and will therefore be integrity-protected to the extent possible.

This document provides guidance for network operation on the management of QUIC traffic. This includes guidance on how to interpret and utilize information that is exposed by QUIC to the network as well as explaining requirement and assumptions that the QUIC protocol design takes toward the expected network treatment. It also discusses how common network management practices will be impacted by QUIC.

Of course, network management is not a one-size-fits-all endeavour: practices considered necessary or even mandatory within enterprise networks with certain compliance requirements, for example, would be impermissible on other networks without those requirements. This document therefore does not make any specific recommendations as to which practices should or should not be applied; for each practice, it describes what is and is not possible with the QUIC transport protocol as defined.

QUIC is at the moment very much a moving target. This document refers the state of the QUIC working group drafs as well as to changes under discussion, via issues and pull requests in GitHub current as of the time of writing.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It’s not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. Features of the QUIC Wire Image

In this section, we discusses those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. Here, we are concerned primarily with QUIC’s unencrypted wire image, which we define as the information available in the packet header in each QUIC packet, and the dynamics of that information. Since QUIC is a versioned protocol, everything about the header format can change except the mechanism by which a receiver can determine whether and where a version number is present, and the meaning of the fields used in the version negotiation process. This document is focused on the protocol as presently defined in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls], and will change to track those documents.

2.1. QUIC Packet Header Structure

The QUIC packet header is under active development; see section 5 of [I-D.ietf-quic-transport] for the present header structure, and https://github.com/quicwg/base-drafts/pull/361 for one current proposed redesign.

Currently the first bit of the QUIC header indicates the present of a long header that exposed more information than the short. The long header is typically used during connection start or for other control processes while the short header will be used on mostly packets to limited unnecessary header overhead. The following information may be exposed in the packet header:

- version number: The version number is present during version negotiation.

- connection ID: The connection ID identifies the connection associated with a QUIC packet, for load-balancing and NAT rebinding purposes; see Section 2.3.

- packet number: Every packet has an associated packet number; this packet number increases with each packet, and the least-significant bits of the packet number are present on each packet; see Section 2.4.

- public reset indication: Public reset packets expose the fact that a connection is being torn down to devices along the path. The applicability of public reset is currently under discussion; see https://github.com/quicwg/base-drafts/issues/353 and https://github.com/quicwg/base-drafts/pull/20.

- key phase: To support 0-RTT session establishment, QUIC uses two key phases; the key phase of each packet must be exposed to support efficient reception.

- additional flags: Additional flags for diagnostic use are also under consideration; see https://github.com/quicwg/base-drafts/issues/279.

[Editor’s note: also further discuss which bits cannot change with versioning]

2.2. Integrity Protection of the Wire Image

As soon as the cryptograhic context is established, all information in the QUIC header, including that exposed in the packet header, is integrity protected. Therefore, devices on path MUST NOT change QUIC packet headers, as alteration of header information would cause packet drop due to a failed integrity check at the receiver.

2.3. Connection ID and Rebinding

The connection ID in the QUIC packer header is used to allow routing of QUIC packets at load balancers on other than five-tuple information, ensuring that related flows are appropriately balanced together; and to allow rebinding of a connection after one of the endpoint’s addresses changes - usually the client’s, in the case of the HTTP binding. The connection ID is proposed by the server during connection establishment. A flow might change one of its IP addresses but keep the same connection ID, as noted in Section 2.1, and the connection ID may change during a connection as well; see

section 6.3 of [I-D.ietf-quic-transport]. See also https://github.com/quicwg/base-drafts/issues/349 for ongoing discussion of the Connection ID.

2.4. Packet Numbers

The packet number field is always present in the QUIC packet header. The packet number exposes the least significant 32, 16, or 8 bits of an internal packet counter per flow direction that increments with each packet sent. This packet counter is initialized with a random 31-bit initial value at the start of a connection.

Unlike TCP sequence numbers, this packet number increases with every packet, including those containing only acknowledgment or other control information. Indeed, whether a packet contains user data or only control information is intentionally left unexposed to the network.

While loss detection in QUIC is based on packet numbers, congestion control by default provides richer information than vanilla TCP does. Especially, QUIC does not rely on duplicated ACKs, making it more tolerant of packet re-ordering.

2.5. Greasing

[Editor’s note: say something about greasing if added to the transport draft]

3. Specific Network Management Tasks

In this section, we address specific network management and measurement techniques and how QUIC’s design impacts them.

3.1. Stateful Treatment of QUIC Traffic

Stateful network devices such as firewalls use exposed header information to support state setup and tear-down. [I-D.trammell-plus-statefulness] provides a general model for in-network state management on these devices, independent of transport protocol. Features already present in QUIC may be used for state maintenance in this model. Here, there are two important goals: distinguishing valid QUIC connection establishment from other traffic, in order to establish state; and determining the end of a QUIC connection, in order to tear that state down.

1-RTT connection establishment, using a TLS handshake on stream 0, is detectable using heuristics similar to those used to detect TLS over TCP. 0-RTT connection establishment, however, provides no particular

heuristic for differentiation from random background traffic at this time.

Exposure of connection shutdown is currently under discussion; see https://github.com/quicwg/base-drafts/issues/353 and https://github.com/quicwg/base-drafts/pull/20.

3.2. Measurement of QUIC Traffic

Passive measurement of TCP performance parameters is commonly deployed in access and enterprise networks to aid troubleshooting and performance monitoring without requiring the generation of active measurement traffic.

The presence of packet numbers on most QUIC packets allows the trivial one-sided estimation of packet loss and reordering between the sender and a given observation point. However, since retransmissions are not identifiable as such, loss between an observation point and the receiver cannot be reliably estimated.

The lack of any acknowledgement information or timestamping information in the QUIC packet header makes running passive estimation of latency via round trip time (RTT) impossible. RTT can only be measured at connection establishment time, and only when 1-RTT establishment is used.

Note that adding packet number echo (as in https://github.com/quicwg/base-drafts/pull/367 or https://github.com/quicwg/base-drafts/pull/368) to the public header would allow passive RTT measurement at on-path observation points. For efficiency purposes, this packet number echo need not be carried on every packet, and could be made optional, allowing endpoints to make a measurability/efficiency tradeoff; see section 4 of [IPIM]. Note further that this facility would have significantly better measurability characteristics than sequence-acknowledgement-based RTT measurement currently available in TCP on typical asymmetric flows, as adequate samples will be available in both directions, and packet number echo would be decoupled from the underlying acknowledgment machinery; see e.g. [Ding2015]

Note in-network devices can inspect and correlate connection IDs for partial tracking of mobility events.

3.3. DDoS Detection and Mitigation

For enterprises and network operators one of the biggest management challenges is dealing with Distributed Denial of Service (DDoS) attacks. Some network operators offer Security as a Service (SaaS)

solutions that detect attacks by monitoring, analyzing and filtering traffic. These approaches generally utilize network flow data [RFC7011]. If any flows pose a threat, usually they are routed to a "scrubbing environment" where the traffic is filtered, allowing the remaining "good" traffic to continue to the customer environment.

This type of DDoS mitigation is fundamentally based on tracking state for flows (see Section 3.1) that have receiver confirmation and a proof of return-routability, and classifying flows as legitimate or DoS traffic. The QUIC packet header currently does not support an explicit mechanism to easily distinguish legitimate QUIC traffic from other UDP traffic. However, the first packet in a QUIC connection will usually be a client cleartext packet with a version field and a connection ID. This can be used to identify the first packet of the connection (also see https://github.com/quicwg/base-drafts/issues/185).

If the QUIC handshake was not observed by the defense system, the connection ID can be used as a confirmation signal as per [I-D.trammell-plus-statefulness]. In this case, similar as for all in-network functions that rely on the connection ID, a defense system can only rely on this signal for known QUIC's versions and if the connection ID is present (also see https://github.com/quicwg/base-drafts/issues/293).

Further, the use of a connection ID to support connection migration renders 5-tuple based filtering insufficient, and requires more state to be maintained by DDoS defense systems. However, it is questionable if connection migrations needs to be supported in a DDOS attack or if a defense system might simply rely on the fast resumption mechanism provided by QUIC. This problem is also related to these issues under discussion: https://github.com/quicwg/base-drafts/issues/203 and https://github.com/quicwg/base-drafts/issues/349

3.4. QoS support and ECMP

QUIC does not provide any additional information on requirements on Quality of Service (QoS) provided from the network. QUIC assumes that all packets with the same 5-tuple {dest addr, source addr, protocol, dest port, source port} will receive similar network treatment. That means all stream that are multiplexed over the same QUIC connection require the same network treatment and are handled by the same congestion controller. If differential network treatment is desired, multiple QUIC connection to the same server might be used, given that establishing a new connection using 0-RTT support is cheap and fast.

QoS mechanisms in the network MAY also use the connection ID for service differentiation as usually a change of connection ID is bind to a change of address which anyway is likely to lead to a re-route on a different path with different network characteristics.

Given that QUIC is more tolerant of packet re-ordering than TCP (see Section 2.4), Equal-cost multi-path routing (ECMP) does not necessarily need to be flow based. However, 5-tuple (plus eventually connection ID if present) matching is still beneficial for QoS given all packets are handled by the same congestion controller.

3.5. Load balancing

[Editor's note: explain how this works as soon as we have decided who chooses the connection ID and when to set it. Related to https://github.com/quicwg/base-drafts/issues/349]

4. IANA Considerations

This document has no actions for IANA.

5. Security Considerations

Supporting manageability of QUIC traffic inherently involves tradeoffs with the confidentiality of QUIC's control information; this entire document is therefore security-relevant.

Some of the properties of the QUIC header used in network management are irrelevant to application-layer protocol operation and/or user privacy. For example, packet number exposure (and echo, as proposed in this document), as well as connection establishment exposure for 1-RTT establishment, make no additional information about user traffic available to devices on path.

At the other extreme, supporting current traffic classification methods that operate through the deep packet inspection (DPI) of application-layer headers are directly antithetical to QUIC's goal to provide confidentiality to its application-layer protocol(s); in these cases, alternatives must be found.

6. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

7. References

7.1. Normative References

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[Ding2015]
Ding, H. and M. Rabinovich, "TCP Stretch Acknowledgments and Timestamps - Findings and Impliciations for Passive RTT Measurement (ACM Computer Communication Review)", July 2015.

[draft-kuehlewind-quic-applicability]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", March 2017.

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-02 (work in progress), December 2016.

[IPIM] Allman, M., Beverly, R., and B. Trammell, "In-Protocol Internet Measurement (arXiv preprint 1612.02902)", December 2016.

[RFC7011] Claise, B., Ed., Trammell, B., Ed., and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information", STD 77, RFC 7011, DOI 10.17487/RFC7011, September 2013, <http://www.rfc-editor.org/info/rfc7011>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Dan Druta
AT&T

Email: dd5826@att.com